Complexity Analysis For Performance Modelling

Komplexitätsanalyse zur Performanz Modellierung
Bachelor-Thesis von Johannes Wehrstein
Tag der Einreichung: 08.10.2019

1. Gutachten: Prof. Felix Wolf
2. Gutachten: Marcus Ritter

TECHNISCHE
UNIVERSITÄT
DARMSTADT

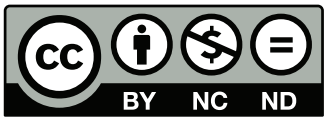
Department of Computer Science
Laboratory for Parallel Programming

Complexity Analysis For Performance Modelling
Komplexitätsanalyse zur Performanz Modellierung

Vorgelegte Bachelor-Thesis von Johannes Wehrstein

1. Gutachten: Prof. Felix Wolf
2. Gutachten: Marcus Ritter

Tag der Einreichung: 08.10.2019



Erklärung zur Bachelor-Thesis gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, die vorliegende Bachelor-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den October 31, 2019

(J. Wehrstein)

Abstract

Current automatic and empirical performance modelling approaches are heavily challenged by large cluster programs. Especially programs with multiple performance relevant parameters are solvable only with high effort, due to the large search space of performance functions, spanned by combining the performance relevant parameters with simple arithmetic operations. The search space is, therefore, increasing extensively with more parameters. Current empirical performance modelling tools like ExtraP are struggling with large search spaces but are able to deal with them. Actually, ExtraP limits its search space to simple functions, which were covering most of the complexity functions of real-world programs, excluding quadratic or cubic functions, to downsize the search space and decrease the modelling time. To overcome the problem of exploding function search spaces, this work evaluates the usage of Deep Neural Networks to predict a rough complexity class of the performance function and therefore enables the option to significantly refine the performance modeller's search space while also covering more function types. The deep learning models are trained and evaluated on synthetic datasets with two and three parameters e.g. amount of processors and problem size. Further, this work introduces a multi-parameter approach, which utilizes pre-trained models dealing with fewer parameters, to support the higher parameter model. Evaluation of the deep learning models reaches an accuracy of 98.6% for predicting the correct complexity class of performance functions with 2 performance relevant parameters and 86% with 3 parameters.

Contents

1 Introduction

1.1 Motivation

A fundamental for developing and deploying parallel programs on computing clusters is the analyzation of this program's metrics especially their runtime performance. Estimating a performance model for a parallel program leads to a better understanding of the algorithmic insights of the program and therefore enhances the ability to improve the performance. Especially performance functions, which are describing the program's runtime based on predefined problem parameters, are crucial. Performance model generation can be classified into analytical and empirical. The first one uses analyzation of the program code, the loops and execution paths, and generates the model from the programmer's perspective like done by Yang et al. [1] and Wang et al. [2]. The empirical approach is based on performance measurements for different configurations of the program like problem size or the number of used processors and was used e.g. by Calatoiu et al. [3] and Shudler et al.[4] for predicting performance. Empiric performance modelling approaches treat the program as a black-box and thereby knowledge about the underlying code is not required. Empirical approaches require no domain knowledge about the problem or the implemented algorithmic solution and can be executed automatically with less supervision. In contrast to analytical approaches, which are way more expensive and often infeasible to model whole programs, empirical performance modelling enables modelling with fewer costs and dealing also with whole and complex programs.

1.2 State of the Art

Existing empiric solutions like ExtraP [5] model performance functions based on various measurements for multiple performance-relevant parameters. Unfortunately, these models are limited to a small size of parameters because with an increasing amount, they are suffering from an explosion of the performance function search space. A search space reduction as much as possible is required. Machine learning-based classification techniques can possibly solve this problem by predicting the basic complexity of the program's performance function before starting the complete modelling process.

1.3 Approach & Methods

Deep learning, a subset of machine learning, looks promising for solving this task because of its opportunity to even solve highly complex problems as shown by Schmidhuber et al. [6] in general and Krizhevsky et al. for Deep Convolutional Neural Networks [7]. So, this work will take a look at the ability of deep learning models to solve this task. The deep learning models were implemented and evaluated using TensorFlow, a recommended tool for large scale machine learning [8], and Keras. Data for learning and evaluating consists of runtime measurements taken on predefined and grid aligned coordinates and the complexity class of the underlying function as labels.
In general, the measurement position is limited to a maximum of 5 points per axis, this means in a two-dimensional space there is a maximum of 25 grid aligned measurements usable as inputs to the machine learning model. The implemented models are running on this grid aligned measurements and the predicted complexity classes are chosen out of a restricted set of classes.

The complexity functions were formed based on these 5 basic complexity classes:

$$O(1), O(log(n)), O(n), O(n^2), O(n^3) \tag{1.1}$$

and containing only two added terms, each one formed by the multiplication of the previously listed base complexities of each parameter. This means in a two-dimensional space the functions are having the following form:

$$f(x,y) = c_1 + c_2 * g(x) * h(y) + c_3 * i(x) * j(y) \tag{1.2}$$

with f, g, h, i as a basic complexity class. This form results in a broad but still precise range of classes. The functions of the

training data can be generated synthetically and were automatically labeled with their corresponding complexity class as further described in chapter 3. The evaluation is done by monitoring the training process and testing on separated datasets by using the evaluation metrics accuracy and f1-score.

The evaluation of the promising approaches is focused on the applicability in a two-parameter and three-parameter function space.

Training, evaluation during training and testing is performed on three different datasets. To ensure comparability, each of the implemented two-parameter models uses the same datasets. Likewise, the three-parameter models were also sharing the same datasets. Monitoring the training process is done by using a check pointer, which evaluates the accuracy of each model on a separate evaluation dataset after each epoch. These results were visualized with TensorBoard. During training the check pointer stores the model state that achieves the highest accuracy during the whole training process. After finishing training, the stored model state with the best accuracy will be loaded and used for evaluating the model. This will ensure that the evaluation of the model is done on the best model state found during training and not on the final one. The trained models were stored afterwards to allow a later on evaluation and specific comparison.

1.4 Contributions

Expected benefits of this work are the following:

- Faster and cheaper performance modelling: using a machine learning approach to predict a rough complexity class of the performance function leads to a significant search space reduction for state-of-the-art performance modelling approaches. This reduction will enable state of the art performance modelling approaches to find the correct performance models significantly faster and with less effort.
- Generation of more accurate model: Current performance modelling approaches are limited in the function space, with the aim to downsize the performance function search space. This machine-learning-based approach is not limited in function space and therefore a machine-learning-based model that predicts the complexity class of a performance function and which includes a broader range of function types, including exponential functions with low exponents, will enable state-of-the-art models to cover a broader range of functions and generate therefore more accurate models.
- Starting point for machine learning in performance modelling and models of all kinds of performance functions: This work proposes the use of a machine-learning-based model to accelerate performance model generation and to make performance functions more precise. Therefore, this work could be a starting point for using machine learning in general for performance modelling. Besides, the proposed approach is able to deal with a broader range of performance function types compared with state-of-the-art approaches like ExtraP. Introducing an approach that deals with a significant broader function space might be a starting point to models dealing with all kinds of performance functions.

1.5 Outline

The thesis is structured as follows. In the following Chapter fundamentals of machine learning especially deep learning and an existing heuristical approach called ExtraP [5], a state-of-the-art empirical performance modeller, will be outlined. An overview of the approach for two parameters, the input and output data and the approach for three parameters is described afterwards in Chapter 3. The overall evaluation of the model, the optimizations, configurations, elaboration of their limitations and reducing the number of measurements to solve the task is done in Chapter 4. Finally, in Chapter 5 the conclusion and possible future works is described. Relevant appendices can be found in the final Chapter 6.

2 Machine Learning and Performance Modelling Fundamentals

2.1 Performance Modelling

Optimization of large programs requires knowledge about the insights and design of the program. Therefore, efficient exploration, especially on large clusters, is required, to lose only less power on performance experiments. Analyzing programs manually is an option for small programs but complex for larger ones. At this point, automatic performance modelers are used taking performance-relevant parameters into account to model a specific behavior of the program, for example, its runtime. Taking these results into account can afterwards improve the configuration and enable further optimizations.

In 2013 Calotoiu et al. [3] proposed an approach to automatically generate performance models for finding scalability bugs in complex code, that allows to model not just only a subset of program. Instead, it allows generating empirical performance models for each part of a program, while also speeding up the generation process. Further solutions like ExtraP[5] are based on this work and uses besides heuristics to model the performance and find a function describing the program's performance best based on performance relevant variables. For describing the performance, a parallel-model-normal-form (PMNF) was introduced by Calotoiu et al. [9] which describes a standard format for performance functions:

$$f(x_1, ..., x_m) = \sum_{k=1}^{n} c_k * \prod_{l=1}^{m} x_l^{i_{kl}} * log_2^{j_{kl}}(x_l) \tag{2.1}$$

with often $n = 2, i \in \{\frac{0}{4}, \frac{1}{4}, ..., \frac{12}{4}\}, j \in \{0, 1, 2\}$. ExtraP will try based on this normal form to find parameters i_{kl} and j_{kl} so that the performance function behaves similar to the real program. Analytical determination of these parameters is computationally expensive because the search space is very large and explodes with a larger n. Therefore, heuristics are used to speed up the modelling process. In 2017 Reisert et al. [10] reference following the blind seer) proposed a new ExtraP approach for automatically determining the coefficients by using regression. Their approach is based on a simplified PMNF containing only two coefficients.

$$f(x) = c_0 + c_1 * x^{\alpha} * log_2^{\beta}(x) \text{ with } \alpha < 6 \text{ and } \beta < 3 \tag{2.2}$$

They set up multiple hypotheses beforehand and automatically determining the coefficients c_0 and c_1 by regression for each hypothesis. For adjusting the coefficients, they do a recursive refinement of the interval. Each refinement step more than halves the coefficient search space. Afterwards, the hypothesis with the smallest error will be chosen. This process can yield good results but is limited to functions expressible by the used normal form. In addition, ExtraP is currently not able to handle multiple parameters efficiently. Therefore, machine learning approaches might be able to advise the ExtraP algorithms by predicting a rough complexity class and thereby reducing the coefficient search space significantly.

2.2 Machine Learning

Machine Learning has become a popular method for solving a problem in recent years. In comparison to analytical problem solving, where a human is used for finding a problem solution, machine learning lets the program finding a solution itself. Therefore, the model gets some data and the expected output and tries to find a solution so that assigning an output for every given input is successful. This kind of method is applied often for tasks were an explicit solution is difficult to find or too much data is involved.

Tasks, where machine learning approaches are applied often, are classification and regression tasks. Classification means labeling data with a single element of a set of classes. The possible labels also called classes were defined specifically for each task. Usual methods for classification are rule learners, which try to find multiple rules describing best

when which label is assigned to the data, or decision trees, a method wherein each node a condition is evaluated and depending on the outcome the path is chosen through the tree. Finally, the leaves are denoting the labels. These two kinds of methods are a good option if the task is not too complex and some values of the data are chosen from a set of options. In general rule learners and decision, trees are simple to understand and interpret but they are inaccurate on many tasks.

Besides classification tasks, there are also various other fields machine learning is applied to, for example, regression. Regression is a task that comes along without a set of enumerable outcomes, but instead, it has as output, for example, a number. This is successfully used for the configuration and optimization of systems. Regression tasks can also be solved with rule learners or decision trees, only by replacing the labels with ranges. In recent years, a method already known for centuries, is used more and more in the field of machine learning which is able to solve such problems very well: Neural Networks. This kind of solution contains many nodes connected with unidirectional edges forming a net together[11]. Each node computes an output value based on its inputs. The computed output is then forwarded to all nodes which were connected by an outgoing edge, until it reaches an output node where no further nodes are connected afterwards. The value computed right there is treated as output. The net has some nodes called input nodes, which will take one value from the data and pass it to the network. For getting good results the computations done in each individual node has to be well chosen. Therefore, a process called training, similar to the process in a rule learner or decision tree learner, tries to configure each node's computation so, that the outcome of the net is equal to the expected value. This is done by passing the measured difference between the predicted outcome and the expected outcome back through the model, so that each node can adjust its values accordingly. For making a smooth adjustment of the computations possible, the computations in each node have a special form:

$$f(x) = \sigma(\sum_i w_i * x_i + b) \tag{2.3}$$

In this formula, x denotes the inputs given to this node, one input from each ingoing edge. These input values are multiplied with a set of node-specific values w called weights and are added with a node-specific bias value b. Afterwards, a non-linear function σ is applied, called activation function. Using a non-linear function here is crucial because computing a linear function on the output of linear functions can be collapsed into a single linear function. But solving a problem with a linear function is mostly not possible, therefore non-linear functions are used like discussed by Ramachandran et al.[12]. The node-specific values, the weights and the bias of each node, are the only adjustable parameters in the whole net. For fitting these values to the problem, the model observes its current output and compares it with an expected output on a training dataset. Based on the difference between these values each node gets an error assigned, estimated by a process called backpropagation[13]. On these errors, computed individually for each node and weight, the weights and the bias are adjusted. This process is repeated until the weights are all well-chosen and the network output is similar to the correct results. This process of computing errors on each node and updating the weights, by taking gradients and momentum into account, is made by the optimizer and can be done in various ways. Therefore, multiple optimizers exist, which have to be chosen for the specific problem. In general, good optimizers are Adam[14] and SGD[15], but each optimizer works better with some tasks and with some not, therefore general advice for an optimizer is not possible. Networks following the described principles and having a minimum of one intermediate layer between the input- and the output layer were also called as multi-layer perceptron (MLP).

Networks, as previously described, are able to solve very complex problems. The nodes inside the net are grouped as layers and only nodes of sequent layers are connected by unidirectional edges. If each node of a layer is connected to each node of the sequent layer, then it's called a dense layer. Connecting all the nodes of the previous and the following layer requires more computational effort but is often used in neural networks. Depending on the problem, the net gets several layers with an individual number of nodes. Nowadays, these nets have many layers, a deep nesting of them, therefore the whole approach is called deep learning.

So long so good, these deep neural networks can solve problems very well as shown by LeCun et al. [16], but the training requires a lot of computing effort and huge datasets. Using huge datasets is crucial because the network should generalize the data, it should find a solution for the problem. Instead, with not large enough datasets, the network will memorize the data instead of generalizing it. This is called overfitting and if this occurs the model is only able to predict good results if the data was covered in the set used for training. If not, the outcome is not that good. Therefore, a combination of small models and large datasets is required. In the previously described machine learning algorithms, they are all learning by comparing the computed outcome with an expected outcome. This is called supervised learning. Assigning to input data a correct output is often very costly. Then, an often-applied alternative is unsupervised learning. Unsupervised learning means, that only the input data is given, but no expected outputs. Such models are way more

complicated but if they can be applied to a task, even problems where no output data is available can be solved. Varying input data is still required for training but labeled data, often produced with large effort and high costs, not. Unsupervised learning is often used in fields of NLP. Due to the non-existent data scarcity not the first choice in this work.

Previous work on using machine learning to assign complexity classes to function values or function interpolation, in general, was done in 2000 by Anthony et al. [17], but there is no recent work on using current machine learning approaches like deep neural networks.

2.3 Performance Function Generation

Machine learning requires large datasets with high quality. The generation of such data has, therefore, a significant impact on the results because if data generation results in the inferiority of the data, the model's outcome will probably achieve less good results. Marcus Ritter proposed a method to successfully generate large and well-distributed datasets. His work is used as a code base for the data generator here. Ritter proposed an approach with generates the functions based on an extended version of the simplified PMNF. The coefficients c_0 and c_1 are chosen in a range from 0.001 to 1000 and are uniformly distributed. Also, the values of α and β are randomly chosen and uniformly distributed. Further strategies like noise and term contribution, described in chapter 4.2 - Generating Synthetic Data, is also based on his work.

3 Approach

Approaches for solving this task can vary between multiple fields, analytic or heuristic approaches like used in ExtraP or also machine learning-based approaches. The field of machine learning approaches includes multiple opportunities where deep learning is as discussed in the introduction the most promising one. Deap Learning has become very popular in recent years due to its property to solve even very big tasks and deal with the complexity and variety of possible outputs very well. AlphaGo[18] is herefore a very popular example, due to its usage of deep learning and its outstanding results in playing the board game go. Different machine approaches like rule-based systems, decision trees or nearest neighbor approaches look applicable but mostly these approaches were used for simpler problems, task with small label space or tasks with limited training data. None of these apply on our problem, therefore deep learning will be the preferred solution for this work. The scope of this work is hereby on deep learning-based solutions.

3.1 Data Inputs

Solving the complexity prediction task can be done as described in chapter 2 by using an analytical approach or an empirical one. Here an empirical approach is used. Therefore, the approach works without information about the underlying program, its task or its implementation strategies. Only information about some measured runtimes is required. The runtimes are measured for varying performance-relevant parameters e.g. problem size or the number of processors the program runs on. The approach comes along with a set of specific parameter configurations, where the measurements should be taken, therefore there are predefined parameter configurations, constant for all programs, where the program's runtime should be measured. These configurations were aligned as a grid as shown in figure 3.1(b) and contain 25 measurement configurations. Further sections will take a look at possibilities to reduce the number of measurements and only take measurement configurations that require fewer resources and exclude configurations with higher effort. The 25 fixed measurement configurations are defined by a set of 5 values for each parameter. The concrete values for the parameters x and y are:

$$\begin{aligned} x &\in \{10, 20, 30, 40, 50\} \\ y &\in \{4, 8, 16, 32, 64\} \end{aligned} \tag{3.1}$$

Equation 3.1: Fixed measurement points for the x and y parameter.

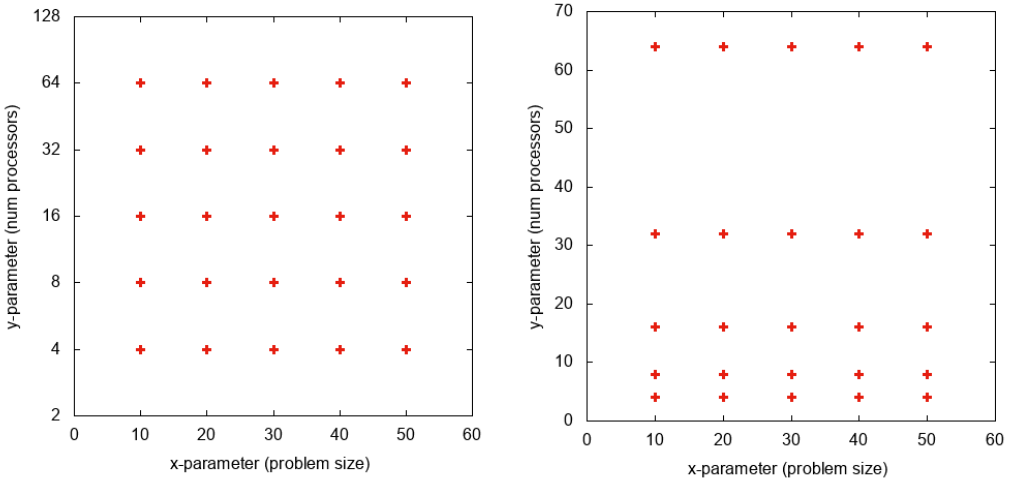
The values of the two parameters were using different underlying paradigms. The x parameter is linear selected. Starting from value 10, the other values are increasing by 10 with each step. In contrast the parameter y doubles in each step, starting from the value of 4. The values are generated exponentially to the base of two. Using this two different paradigms to choose the measurement values will adapt very well to often taken measurements with exponentially increasing processor amounts and a linear increasing problem size and should increase the opportunity to reuse already taken measurements for applying this approach on, or to use the measurements specifically elaborated for this approach with other, for example more precise and later applied, approaches. Another advantage is that taking in addition to a linear value choosing paradigm an exponentially based value chooser, starting on a lower value, instead of taking the same linear value chooser again, the values for the exponential taken parameter were in sum significant lower than the linear taken values in this case, and therefore the amount of effort for measurement computing will lower. Figure 3.1 (b) visualizes that the majority (60%) of the measurements were computed with a y less equal than 20 using the exponential distribution of y-values instead of 40% with the linear aligned values used on the x-axis.

These 5 measurement points on each axis, are used to define the measurement coordinates. The coordinates are formed by the cartesian product of the measurement points for the two axes, which results in 25 configurations formulated in equation 3.2.

For each of these measurement coordinates, the runtime is measured. These 25 runtimes form the input data of this approach. The model tries then to extract a complexity function that can describe the runtimes best.

$$\text{measurements}(p) = \{p(x,y) \mid x \in \{10,20,30,40,50\} \wedge y \in \{4,8,16,32,64\}\} \tag{3.2}$$

Equation 3.2: Measurements of a program p for the configuration parameters x and y.



(a) Visualization with log scaled y-axis (base 2). **(b)** Visualization without scaling.

Figure 3.1: Visualizing the measurement coordinates with an logarithmic scaled y-axis (a) and without scaling (b). The red crosses mark the configurations were the measurements will be taken.

3.2 Outputs

The purpose of the model is to predict the complexity class of the program, by analyzing its runtime measurements. Therefore, the model output should be a complexity class. In general complexity classes are exhaustively researched in a two-dimensional space, this means with only one parameter. But there is less research in multi-dimensional space. Howell et al [19] show that the rules and processes of the two-dimensional space can successfully be adapted to a multidimensional space, with two and more parameters, and are still valid. Based on this work, the complexity classes can be identified by using the Big-O notation. Difference between complexity classes in a two-dimensional space and a multi-dimensional space is the number of possible classes. In higher dimensions more parameter combinations are possible and random generated functions are during complexity class extraction less likely reducible to the same complexity function. Therefore, the predicted complexity functions were limited to be constructible out of these basic sub-functions, called in further chapters basic-complexity-function-building-block.

$$1, log(n), n, n^2, n^3 \tag{3.3}$$

Equation 3.3: Basic functions, which were exclusively used for complexity function construction.

Further the complexity functions may have up to two added terms, each one consisting of a single basic complexity function building block for each parameter, which were multiplied together. In general, the functions will then follow the form:

Further complexities like $O(n^4)$ aren't scope of this work because they were rarely occurring and mainly avoided in programs. Other complexities like $O(n * log(n))$ were also discarded due to their similarity to other basic functions. In this case, it's less helpful to distinguish between $O(n * long(n)$ and $O(n)$ than between $O(n)$ and $O(n^2)$ because, in comparison to the whole covered function space, they have nearby values and nearby gradients.

The function space, which is following previously described restrictions, forms 625 functions with only 125 valid complexity class functions. 500 of the 625 functions following the restrictions do not match conditions for complexity classes in Big-O notation, like they contain redundant terms, or the complexity is determined by only one of the two added terms, so the other term is not relevant for the complexity class and can therefore be ignored and removed. Taking these

$$f(x,y) = g(x) * h(y) + i(x) * j(y) \tag{3.4}$$

Equation 3.4: General form of the predicted complexity class functions with $g(n), h(n), i(n)$ and $j(n)$ as basic complexity function building blocks and x and y as the parameters.

Form	Amount
$const$	1
$g(p_1)$	8
$g(p_1) * h(p_1)$	8
$g(p_1) + h(p_2)$	8
$g(p_1) * h(p_2) + i(p_1)$	56
$g(p_1) * h(p_2) + i(p_1) * j(p_2)$	44

(3.5)

Table 3.1: Overview over the Complexity Class distribution for 2 parameters. The functions are separated into 6 groups based on the form of the function. g, h, i and j denotes the basic complexity function building blocks and p_1 and p_2 two parameters.

properties of complexity classes in Big-O notation into account, the amount of different functions reduces to 125. A separation of these 125 complexity functions and their distribution over these splits are shown in table 3.1. The complete list of predicted complexity classes are appended in section 6.1.

This task, to predict the correct complexity out of a set of complexity functions, is a standard classifications task. Therefore, a multi-layer-perceptron is used for solving. The multi-layer perceptron is a good method to solve complex classification or regression tasks as described in chapter 2.1.

The basic approach, where all the models are based on, takes 25 runtime measurements and predicts the complexity class function. The minimal model contains an input layer and afterwards a softmax activation layer. The input layer has 25 flat aligned input nodes so that each measurement has its own input node. The mapping of the measurements to the input nodes is based on the coordinates. This means that the measurements of different functions at the same coordinate will map to the same input node. Mapping the measurements constantly allows to get along without further metadata. The model has no information about the real measurement coordinates, but this will be easily learned implicitly by the model. Therefore, the constant mapping is crucial, because otherwise metadata about the inputs, the real measurement coordinates, is required. Adding the coordinates to the input data would blow up the model and slow down the training process, because the model has to learn how to use this additional information first, to interpret the measurements, before predicting good results. This learning can be replaced by using a fixed mapping that reduces the training task only to the core complexity class prediction. The chosen mapping is identical to the format the data is generated. The grid aligned measurements of the form 5×5 are flattened to a 1×25 array and can therefore directly be mapped to the input layer. So, the model takes only the position of the measurement in the parameter configuration list into account and no further metadata.

Each model adds as the last layer a densely connected softmax activation layer, in the minimal model directly on top of the input layer. The softmax layer contains as many nodes as labels, for 2 parameters the softmax layer has 125 output nodes, each one corresponds to one complexity class. Using a softmax layer adds a probability distribution to the output, a standard method for classification. Afterwards, the probabilities of the labels can be used to predict the result, hereby using argmax, which returns the label with the highest probability. There are also different opportunities on how to choose the best result based on probability outputs, which takes these probabilities into account. In natural language processing tasks such methods are used often because the label space is mostly very large, often the whole vocabulary space. Therefore, the predicted probabilities are likely smaller than for a smaller number of labels. Another reason for using a different method than argmax is that results might be very similar and therefore the model cannot perfectly distinguish between these labels and assert them a similar output prediction. This task has a not large enough label space, and more important, good distinguishable labels.

To classify a given set of inputs to one of the 125 different complexity classes, the last layer of the model, using the softmax activation function, asserts to each class a probability, which describes how sure the model is that this is the correct class. Afterwards, the class with the highest probability is taken and the model will be evaluated on that. The specific configurations and their impact on the results are evaluated and discussed in chapter 4-Evaluation.

Other approaches like Convolutional Neural Networks (CNN) look not being helpful because CNN models try to find characteristics in subsets of the data. But the task of predicting a complexity class cannot be divided or separated. Therefore, using a CNN is not helpful at this point and not implemented. Later I will discuss this method again for the field of 3 parameter models.

3.3 Approach For 3 Parameters

Previous elaborations were based on tasks with two given parameters. Here, a scenario of three and more parameters will be described.

First of all, the input data is very similar to the 2 parameter inputs. Each parameter still has 5 predefined measurement positions. The measurement positions of x and y are the same as defined in 3.1. The third parameter, named z, has the following predefined measurement positions:

$$z \in \{1, 2, 3, 4, 5\} \tag{3.6}$$

Equation 3.6: Fixed measurement points for the z parameter in a three-parameter scenario.

Choosing small positions - in comparison to x and y - is based on minimizing the effort for measuring the program on the given configuration. Also, it should enable using this approach with small-scaled parameters, were magnitudes of 20 and above are not usual or possible. Two parameter programs, which cannot be measured with high chosen measurement points, is not solvable with the previously described two-parameter approach but indeed with this three-parameter approach, due to the small chosen z parameter. In this case, one of the high chosen parameters x and y will be ignored during measuring and the measurements are duplicated for the ignored parameter. Another reason is explained in the context of the composed three-parameter model later on in this section. In addition, a practical observation, which turns out in a later stage, was, that using larger values for the measurement positions, will end up for the proposed approach in taking way more memory in comparison to measurement positions starting at 1. Storing this datasets, even for 10^6 datasets large ones, is not the problem here, but the deep learning model using this data and holding it in a specific for the approach tailored representation during training, ends up in requiring more memory in the RAM than available in the used machine (memory limitations of GPU server were at 94 GB RAM (64 GB + Swap)). This increase in memory usage seams not worthy and neither promising to reach better results nor widening the application space. Taking memory considerations into account was necessary but not decisive for the decision to use smaller measurement positions for z (memory consumption increasing shows out in a later attempt to improve the results, where the measurement position for z was set equally to x).

The whole amount of measurements is, with 5 measurement positions per parameter, $5^3 = 125$.

For the output classes, the same restrictions as in the two-parameter approach were applied. The functions are formed out of the same basic complexity function building blocks and still consist of two added terms at most. Only the generation of such a term differs, it may now contain three multiplied complexity function building blocks, with one for each parameter.

A widening of the function space, maybe for an additional third added parameter, will increase the label space significantly. Therefore, the decision to restrict the 3-parameter functions only to two added terms is based on observations about possible complexity classes, because investigations on complexity classes in a 3-dimensional space show that the label space explodes with further terms added to the function. It shows out that in a 3-dimensional space, were functions following the above restrictions, 4625 different complexity functions and thereby classification opportunities were possible. This amount of 4625 classes still contains only valid and unique complexity classes with at most two added terms seems challenging for simple deep learning approaches. Without reduction, or with a third added term, the amount will be way larger. Restricting to two terms also agrees with the goal of this work, to predict a rough direction in which the complexity functions goes, and not a deeply detailed function.

The apportionment of the classes is shown in table 3.2.

Form	Amount
$const$	1
0x1	12
0x2	48
0x3	64
1x1	48
1x2	384
1x3	352
2x2	924
2x3	1935
3x3	1145

Table 3.2: Overview over Complexity Class distribution for 3 parameters. The classes were separated by the number of parameters contained by each of the two terms, therefore 1x2 means one term consists of one parameter and the other term of two parameters.

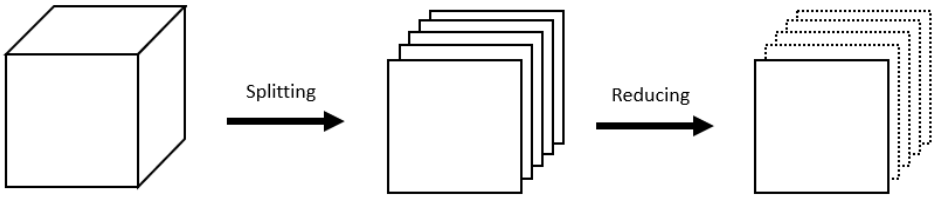


Figure 3.2: Process of first splitting the cube into 5 planes. Afterwards only the first plane is taken, the other ones are discarded.

The input nodes of the model increase to 125, one node for one measurement. Also, the number of nodes of the last layer increases, the softmax activation layer has 4625 nodes, each node corresponds to one class.

The first layer and the last layer of the model were previously described and set by the amount of measurement and the class space. The two-parameter model uses a special configured MLP, but for 3 parameters, different and more options were possible. Implementing the layers in between can still be done by a standard MLP and will be evaluated in chapter 4.4. Also, another strategy has been traced. The initial cube aligned dataset of 125 measurements can be separated into 5 planes as shown in figure 3.2. Looking at this representation raises in mind links to a CNN. These planes look like separate layers, which can be processed separately by the same kernels and the results can be summed up together afterwards. This idea let us decompose the 3-parameter dataset, into only 2 parametrized ones. Information which can be extracted from this data is now the same as in the 2-parameter case. Therefore, the previous model for two parameters can be applied here. So, the 2-parameter model can be applied to each 5x5 measurement plane. Looking further into this method shows that each plane will result in the same complexity function only depending on two parameters, because the planes are only varying in their absolute values, but not in their internal relations. The third parameter is nothing more than a multiplied constant coefficient and thereby not relevant for Big-O complexity classes.

Equation 3.7 shows an example this underlying three-parameter complexity function: $f(x,y,z) = log(x,2) * y^2 * z + x * z^3$. After splitting the model into planes on the z axis the five complexity class functions were listed, each describing one plane and resulting in the same complexity.

Only for terms with a $log(z)$ part the planes show differences between their underlying complexity classes. This difference shows up if the cube is split along the z axis, then the term containing $log(z)$ will get lost for $z = 1$. So, the first plane will likely result in a different complexity than the other planes. Thereby, there is little advantage in computing the two-parameter model on all planes only for catching cases with a $log(z)$ term, instead of just computing it once, but this is negligible. Except for this small exception, there's no loss in information if the classification is only computed on the first plane. The approach will further split along the z axis and take the first plane with $z = 1$. This raises the $log(z)$ term-gets-zero exception but it has a big advantage in contrast to splitting along x or y. The two-parameter approach is trained on data with two parameters. This basic information is relevant for this purpose, because while taking the first plane of the along z split cube, this dataset is inside the training data area of the 2 parameter approach, because the z parameter is now 1 and then there is no difference in the generation of this first plane and a single dataset of the two-parameter approach. This means that the trained two-parameter model can be reused for this task without value shifting

$$
\begin{aligned}
f(x,y,1) &= log(x,2) * y^2 * 1 + x * 1^3 \text{ results in: } compl(f(x,y,1)) = log(x,2) * y^2 + x \\
f(x,y,2) &= log(x,2) * y^2 * 2 + x * 2^3 \text{ results in: } compl(f(x,y,2)) = log(x,2) * y^2 + x \\
f(x,y,3) &= log(x,2) * y^2 * 3 + x * 3^3 \text{ results in: } compl(f(x,y,3)) = log(x,2) * y^2 + x \\
f(x,y,4) &= log(x,2) * y^2 * 4 + x * 4^3 \text{ results in: } compl(f(x,y,4)) = log(x,2) * y^2 + x \\
f(x,y,5) &= log(x,2) * y^2 * 5 + x * 5^3 \text{ results in: } compl(f(x,y,5)) = log(x,2) * y^2 + x
\end{aligned}
\tag{3.7}
$$

Equation 3.7: Example that shows that different planes of a three parameter measurement set still have the same complexity class.

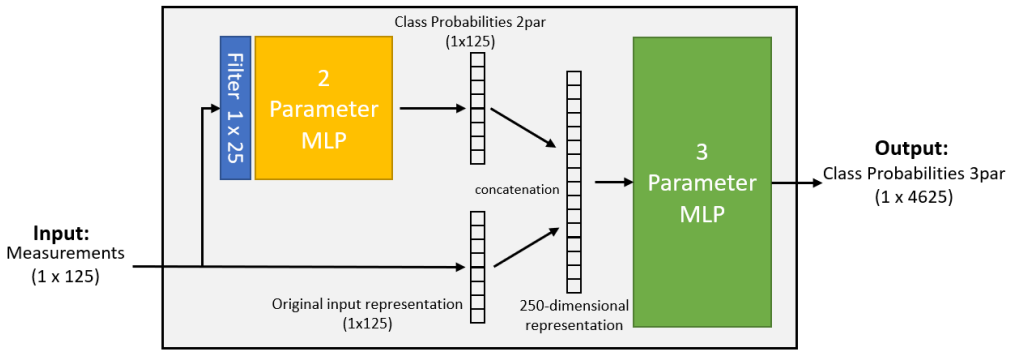


Figure 3.3: Overview over the composed three-parameter approach. First, the pre-trained two-parameter model is applied on a filtered subpart of the data as shown in figure 3.2. Afterwards, the predicted class probabilities will be concatenated with the original input data. The 3 parameter MLP is running then on a dataset with the previously extracted information about the complexity in the single planes in addition.

or further adjustment of the data. The function describing the data of the first plane of the three-parameter dataset split along z has the same probability distribution during data generation than the underlying function describing the two-parameter dataset. This main property leads us to the ability to reuse a two-parameter model, trained on standard two-parameter data, for the three-parameter task without any adjustments.

The further processing of the 2 parameter class probability distribution is shown in figure 3.3. The predicted class-probabilities are concatenated with the original 125-dimensional input vector and then processed by an MLP. The goal of this method is to add more information to the input data so that the three-parameter model can use this additional feature to get a better classification and thereby downsize the modell's size and the trainable parameters because the 2-parameter MLP is still perfectly trained. Finally, the 3-parameter MLP will hopefully reach better results (evaluation in chapter 4.4). This approach is later on called a composed three-parameter model. In addition, the results of the 2 parameter MLP can also be used as a sparse input to the 3 parameter MLP by applying argmax on the 2 parameter MLP output. The input data of the 3-parameter MLP will then have the shape 1x126 instead of 1x250. Using a sparse output instead of a full class distribution will lower the data size but might also lower the ability for the model to use this additional feature. This will be evaluated in section 4.4.2.

4 Evaluation

Configuring deep learning models accordingly to the problems that maximum results are reached is difficult as described by Glorot et al. [20]. Choosing the right solving strategies, tuning configuration parameters and preparing the data well have a heavy impact on the results of the model. Therefore, tuning a parameter and evaluating the decision is required to find a good solution.

4.1 Data

Deep learning requires large datasets for learning a good model. The availability of data that can be used for training is very essential because the complexity of the deep learning model is limited either by computing power or by the available training data. Training a model with an insufficient amount of training data leads to unintended behaviors of the model: the model starts to reproduce the data instead of generalizing it. In the case of function modelling, training-data can be generated synthetically. Real-world data is not necessary, the core training data can be formed out of synthetic functions very well. Further prerequisites for good training are stratified datasets and no bias in the data.

4.1.1 Generating synthetic data

Training deep learning models requires large amounts of data. Often only real-world datasets are available, data-generation is mostly not possible or only sampling new data out of existing data is possible. In the case of performance modelling, synthetic data generation is possible. The data the model uses for learning contains 25 grid-aligned measurements of the runtime for a specific program and machine configurations and the complexity class label the model should predict successfully after training. Therefore, 5 measurement points are chosen for each dimension, so that the measurements for a two-dimensional function were computed at 25 predefined and grid aligned coordinates. The measurement coordinates there defined as following:

$$dataset(f) = \{f(x,y) | x \in \{10,20,30,40,50\} \wedge y \in \{4,8,16,32,64\}\} \tag{4.1}$$

The distribution of the two parameters differs, one is linear, the other exponential. The decision for using different distribution for the two parameters is based on the aim to increase the applicability of the model build later on these datasets. Besides, this measurement configuration can be used for performance measurements with an exponential changing number of processors and a linear changing problem size.

After generating a function, the labels and measurements can be computed from this underlying function. Generating suitable sets of functions is of high importance because all observations that can be made on the labels of the training data, like stratification or bias, are based on the underlying function generator.

Generating functions that are pretty similar to real-word application functions is required to classify the real-world data correctly. If there's somehow a difference between the generated and the real-world dataset, the model might learn a different model and depending on the problem complexity and the used approaches the model is possibly not able to solve the task correctly. Naming a function or a set of functions similar to real-world performance functions underlay some conditions:

- no bias
- same distribution
- similar terms
- Real-world side effects like noise or measurement errors

To generate data similar to real-world data and to overcome some general deep learning issues different strategies were used and, in the following, explained.

To achieve similarity as listed above as much as possible and further not collide the complexity class restrictions, the functions are first of all generated in a form containing, in addition to the parameters and the basic complexity function building blocks, some coefficients these terms were multiplied and shifted with.

The general form of the functions and coefficient ranges are defined in the following way:

$$func(x, y) = c_1 + c_2 * p_1(x) * p_2(y) + c_3 * p_3(x) * p_4(y) \tag{4.2}$$

$$c_1, c_2, c_3 \in \mathbb{R} \wedge c_1, c_2, c_3 \in [0.001, 1000] \tag{4.3}$$

$$p_1(n), p_2(n), p_3(n), p_4(n) \in \{1, log(n), n, n * log(n), n^2, n^3\} \tag{4.4}$$

The multiplicands added to the parameters are used to differentiate the generated functions and to cover possibly all functions with two added terms. A coefficient is added to the whole function to shift the function away from the origin to model runtime of programs which have a constant part which is independent from all varying performance relevant parameters. This can be for example a setup time or time consumptions for scheduling and execution.

On real-world performance function coefficients, there's no known specific distribution, so the coefficients are treated here as uniform distributed and were generated accordingly using a random uniform distribution over the whole range $[0.001, 1000]$. The use of randomness right here is meant to countervail bias production. Bias is a difficult detectable danger. Actually, a model or dataset can be only tested against a bias the supervisor might notice or think of. This means only statements about the absence of a specific bias can be made but statements about the absence of bias, in general, is impossible. No general approaches are yet found. Therefore, randomness is here largely used to overcome this problem.

Noise is everywhere in real-world datasets and measurements and should be also found in this synthetic data. Function terms that contribute less to the function than noise are hard to detect and are not increasing the expressiveness of the function largely. Therefore, at function generation, two hyperparameters are set: the noise and the term-contribution. Noise sets the maximum deviation between the measured and the true values. At the function-evaluation time, where the function is evaluated on the predefined coordinates, adding of noise for each measurement is performed independently to simulate a real-world measurement scenario. Noise is computed by adding a random normal distribution with $sigma = \frac{measurement * noise\_value}{2}$. The noise adding process is executed five times on one measurement for each measurement coordinate and becomes averaged afterwards to on the one hand add noise to data but on the other hand simulate an averaging of results of multiple executions, mostly done for time measurements of programs, to countervail noise in measurement environments. The level of noise for the training data is chosen at 2%. Adding noise to the data possibly leads to the inability to find very small terms. Therefore, the term contribution restricts the function generation, to generate only terms that have a minimum influence on the function values. The minimum percentage of function contribution must be reached both for the function minimum and the function maximum on the set of the measurement coordinates. The coordinates of minima and maxima of a function are the coordinates with minimal parameter configuration (x=10, y=4) and maximal parameter configuration (x=50, y=64) because all the basic complexity building blocks are monotonically increasing, and the coefficients are chosen greater zero. A negative partial gradient can thus never arise. Thus, the function minima can be found by choosing the minimal measurement point for each parameter and the maximum by choosing the maximal measurement point for each parameter. For generating training data these two adjustment screws were set to noise=2% and term contribution=6%, and based Marcus Ritter recent research, adjusted as good and many-scenarios-covering.

A well-investigated requirement for training data is the property of stratification. This enforcement of an equal distribution of labels in the datasets is done here by and enforcement strategy where the filling is as usually done on single class level and enforcement on the level of the label categories defined in table 3.1. More particular enforcement would limit the number of functions that were reduced and generated with a different label intention. The amount of reduced functions is quite large as seen in the numbers of possible two-dimensional functions (625) and complexity classes (125). Using enforcement on the level of single classes would still result in functions that were reduced before added to the dataset. But the amount of them per class increases with later processing, means early processed classes contain only fewer ones because classes processed right at the beginning will likely be filled earlier than other classes - as intended by class enforcement - than classes processed later. This problem can be overcome by using a round-robin approach to fill the classes but the chance to result in bias in the data will somehow or other raise. This impact depends on the processing order and concrete implementation, but here the quality of the data will in no case outperform a stratification with enforcement on the category level, actually, it will lower. This class enforcement on class category level takes the amount of labels contained by a category into account, therefore the probability to generate a function with only a constant term has a probability of $\frac{1}{125}$ while predicting a $f(p_1) * g(p_2) + h(p_1)$ term has a probability of $\frac{56}{125}$.

Generating functions as described above results in way more function generation iterations than the intended amount of functions. The term contribution and the stratification process discard many functions either the term contribution is

not reached, or the class is already filled. This leads to a lack of performance. The amount of successful iterations per second decreases with time because more and more classes reach the maximum amount of functions to get a uniform distribution and won't be added as intended by stratification. Besides that, the term contribution of 6% results in 25 not generatable classes like $x^2 + log(y)$ or $y^3 * x^2 + log(y,2) * x^3$. Therefore, a single run with a uniform distribution of all 125 classes is not possible. Instead, the function generation is done in the following four stages:

1. Random function generation - limited by uniform distribution over all classes
2. Enforced function generation - limited by uniform distribution over all classes
3. Random function generation - limited by uniform distribution over all classes which contain at least one function
4. Enforced function generation - limited by uniform distribution over all classes which contain at least one function

Not predictable classes depend on the chosen term contribution. But this user-chosen parameter is known first at execution. So, removal of unpredictable classes from the set can be done only after a sufficient amount of iterations, because nonpredictable classes are not known before. Removing the set of classes, the distribution is computed on, leads to a higher amount of iterations per second, because the generation takes therewith not predictable classes into account. For further enhancement of performance, a deviation of 10% is added to the maximum amount of class elements defined by the uniform distribution at the fourth-generation stage. The end of the stages is defined by a minimum amount of consecutive iterations which were all generating invalid functions. This threshold should be well chosen because if the threshold is too small, the class enforcement endures not long enough, and some classes will be under- or non-predicted. On the other hand, if the threshold is too high, the classes were filled uniformly but the time for generating a function that still will be added to the dataset increases quadratic, because the chance to generate a valid function and start the unsuccessful iteration count again with it increases with adding more iterations to the threshold. So, this threshold marks the trade-off between runtime and level of uniform distribution. For generating the larger training datasets with a size of 100k or 1m a threshold value of 25% of the final dataset size is chosen. Taking the runtime of the function generation into account is crucial because generating a 100k two-dimensional dataset takes with the recommended configuration 4 hours. A dataset of size 1m takes more than a day.

The strategies come together as shown in listing 4.1.

```
def generateDataset(min_term_contrib, noise):
  while(True):
    c1, c2, c3 = generateRandomCoefficients()
    x1, x2 = generateBasicComplexityBuildingBlocs("x") # generate something like x^2, log(x), ...
    y1, y2 = generateBasicComplexityBuildingBlocs("y") # generate something like y^2, log(y), ...
    term1 = c2 + "*" + x1+"*"+y1
    term2 = c3 + "*" + x2+"*"+y2
    function = c1 + "+" + term1 + "+" + term2

    #check term contribution for min and max configuration for both added terms
    if(min_term_contrib > checkContribAtMin(term1, function)):
        continue
    if(min_term_contrib > checkContribAtMax(term1, function)):
        continue
    if(min_term_contrib > checkContribAtMin(term2, function)):
        continue
    if(min_term_contrib > checkContribAtMax(term2, function)):
        continue

    measurements = evalFunctionWithNoise(function, noise)   # get measurements and add noise
    comp_class = getComplexityOfFunction(function)          # classify function
    return function, comp_class, measurements
```

Listing 4.1: Sample code explaining how synthetic functions and its measurements are generated. First, the function is randomly created. Afterwards, the term contribution will be checked and noise added to the measurements.

4.1.2 Quality of the synthetic data

The generated functions were well distributed and covering all classes except the classes prevented by the term contribution. A noise of at most 2% is covering many measurement environments, but there are systems with clearly more

than 2% noise. For these systems, the model that will be trained on these training datasets won't perform well. But with an increasing noise level, the signal-to-noise ratio gets worse and fewer terms can be identified successfully. Therefore including more parameters in the model is only possible if the further parameter has a higher influence than the noise. Therefore an increase in the level of noise will reduce the parameter identification ability. So with realistic expectations, a noise level of 2% is a good tradeoff between the supported measurement environments and the number of identifiable parameters.

4.1.3 Dataset sizes

Deep learning approaches require large datasets, therefore the dataset size the implemented 2-dimensional approaches are training on is of size 100k while evaluating and testing is performed each on datasets with size 10k. This data set sizes are tested as sufficient, because an overfitting to the training data cannot be seen, not even in the largest models containing more than 20 layers. Due to the stratification of the datasets, the testing datasets are proven as sufficient large, testing on larger datasets deviates negligible with less than 1%.

4.1.4 Generating data with 3 parameters

In a three- or multi-dimensional space the functions are similarly formed. The function is still formed out of two added terms, each one formed by the parameters placed into a basic complexity function building block.

$$func(x,y,z) = f(x) * g(y) * h(z) + i(x) * j(y) * k(z) \tag{4.5}$$

The decision to restrict the 3-parameter functions only to two added terms is based on observations about possible complexity classes. Investigations about complexity classes in a 3-dimensional space show that the label space explodes with further terms added to the function. It shows out that in a 3-dimensional space were functions following above scheme 4625 complexity classes were possible. This amount contains only valid and unique complexity classes, without the reduction the amount will be way larger. The apportionment of the classes is already shown in table 3.2. This explosion of possible class requires further refinement of the function generation process. The sizes of previously generated datasets for the 2-dimensional space were not sufficient for a 3-dimensional model, hence, the training dataset for the 3-dim approaches contain 1m functions. This increase in classes makes it nearly impossible to get a uniform distribution over all possible classes due to the time consumes for generating functions. Applying the methods already used for the 2 parameter functions and changing the threshold to 10% of the final number of datasets that should be generated reduces the generation time to 40 hours (single thread execution with 25% deviation from uniform distribution at most per class).

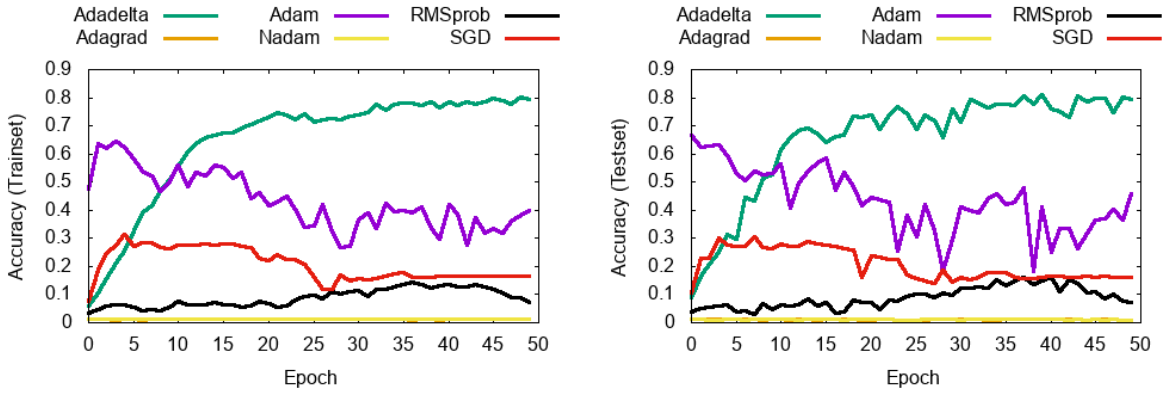
Applying noise and term contribution also to the 3 parameter data sets downsizes the number of different complexity classes in the 1m dataset to 1970 (see 3dim_1m_6-2 dataset).

4.2 Approach Evaluation

4.2.1 Learning configurations (optimizer, batch size, loss function)

While defining the model there were various options to configure the model and the training process. Choosing the right optimizer is very important. In general, some optimizers were said as increasing accuracy very fast but finding only suboptimal solutions e.g. Adam, while other optimizers like stochastic gradient descent (SGD) are usually described as slow but find a local optimum very well. Mostly there are multiple applicable optimizers that can be used for solving a problem only differing in runtime and less in accuracy. But it shows out that for this task only the optimizer AdaDelta is suitable. Applying AdaGrad on a naive and simple model directly reaches an accuracy of 79% while neither Adam, Adagrad, RMSProb nor Nadam reaches 60% (Figure 4.1). This was proven during experiments 28, 29, 32-34, 36 by taking an untrained model, a 100k dataset for training and a 10k dataset for testing, while only replacing the optimizer for each experiment. The comparability of the results is given because the models were trained on the same data while also the influence of randomness to the training process is reduced by that.

As shown in Figure 4.1 Adadelta reaches the highest accuracy and shows an excepted learning curve. During the first 15 epochs, there is a linear and continuous increase in accuracy. In the following 35 epochs, the accuracy improvement slows down and converges to 78%. In contrast, the optimizer Adam shows a way faster improvement of accuracy and reaches after 3 epochs an accuracy of 62%. But the expected behavior of converging or jiggling around this maximum



(a) Accuracy on train dataset.

(b) Accuracy on test set.

Figure 4.1: Comparing different optimizers on the same model and datasets

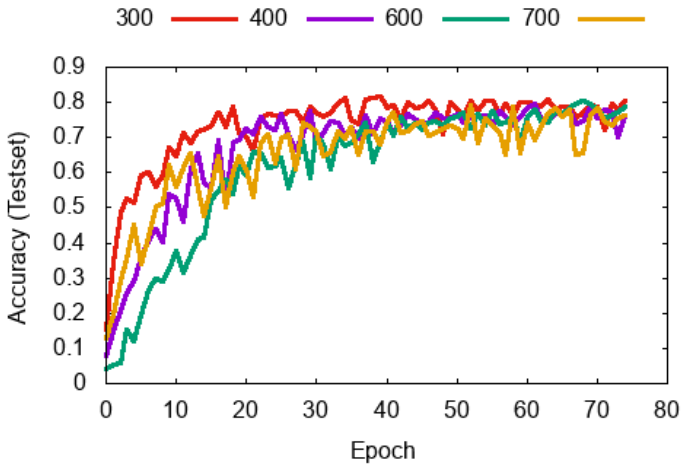


Figure 4.2: Evaluation of batch sizes for values between 300 and 700 for the optimizer AdaDelta.

accuracy is not shown. Contrariwise the accuracy lowers significantly and reduces to about 40% on the test set. These observations make it clear that the Adam optimizer isn't able to continuously push the accuracy up to a level of 90% and above. The other optimizers SGD, RMSProb, Adagrad and Nadam were in contrast to Adam and Adadelta not able to push the accuracy to more than 30%. Based on these observations the further models were all using the Adadelta optimizer, to achieve a consistent and long-lasting improvement of accuracy.

It seems that only AdaDelta solves the problem well because it takes not the whole past history of gradients and momentum into account. Instead, it uses a smaller window.

Other hyperparameters didn't showing such an impact like the optimizer and were not that crucial for the ability to solve the task but still have an impact on the accuracy and mainly in the runtime of the training process. Varying the batch size shows that using a size of 300 results in the highest accuracy, fast learning and a justifiable runtime. Higher batch sizes still reduce the time for one epoch but were generalizing not as well as with batch size 300 and reaching an accuracy between 76% and 79% while using a batch size of 300 reaches 81% (Figure 4.2).

The task is a classification task with more than a dozen classes. Therefore, it's advised to use the categorical cross-entropy for calculating the loss. This function is well suited for classification and also for calculating a helpful loss for many classes. Different loss functions like mean squared error, binary cross-entropy or hinge loss are not fitting as well to this task as categorical cross-entropy.

A simple implementation with only 1 hidden layer, which contains 200 nodes, already reaches an accuracy of 72% after 15 epochs on the standard 100k dataset. This level of accuracy produced by this simple model shows that the problem is in general well solvable with deep learning methods. Differentiating between many classes can be done very well, with

FT Optimizer	Accuracy
10	0.7828
15	0.8216
20	0.7076

Table 4.1: Experimenting with different amounts of layers. It shows out that increasing the number of layers is good until some point. Then the accuracy will drop.

less effort.

4.2.2 Accuracy raising strategies

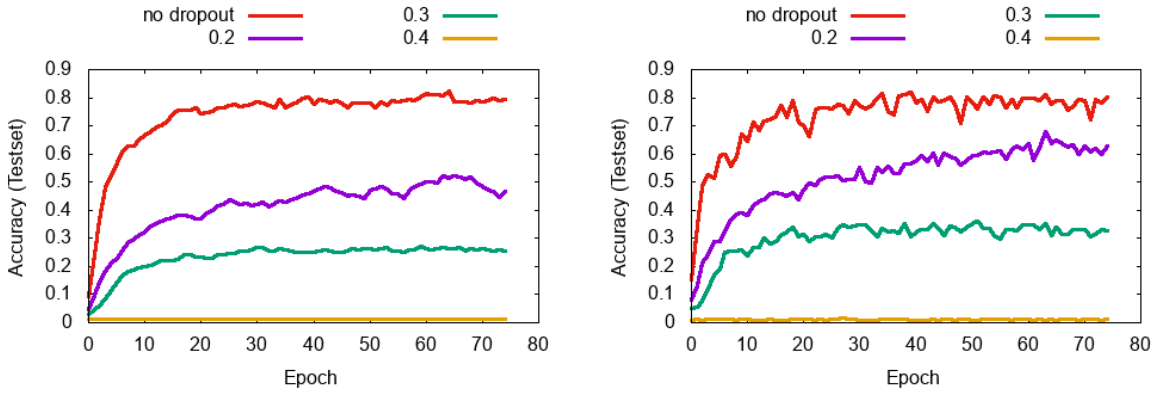
In the following, some strategies to raise accuracy were discussed. The tested strategies were the following:

- Deeper Network (adding further layers)
- Change the shape of the layers
- Dropout
- Larger Datasets
- Kernel Regularization
- Finetuning

A model with only one hidden layer is able to reach an accuracy of 71.79 percent (experiment 235). Much better results are not possible with this shallow model. Common strategies to let an MLP solve more difficult problems, and handle more complexity is to add more layers. By using 20 layers, each one with 200 nodes, does not increase the accuracy, its best accuracy is 70.7%. But with 15 layers there is a significant increase. Adding more layers raises significantly the accuracy (see table 4.1), but also has drawbacks. Due to more layers, there are more trainable parameters, so the training process will slow down and the memory size increases. In addition, using too many layers results in problems with the gradients. The errors measured in the last layer during training can only be propagated well to the previous nodes if they are well initialized. Experimenting with the number of layers shows that with fewer layers, for example, 10 or 15, a significantly higher accuracy can be achieved (experiment 42,43 and 234) in comparison to both the one-layer model and the 20-layer model.
The second applied strategy is to change the shape of the layers. For networks, a well-chosen layer shape is essential, because if the layer contains fewer nodes, it cannot handle the complexity of the problem as well or take all available information from the data into account. Choosing the layer dimension too high slows down the training process and the layer will extract besides nonexistent features out of the data, which will have a bad influence on the results until the following layers learn to ignore these nodes outputs. In addition, using too many nodes lowers the level of generalization because the model will be able to take more details into account and might overfit the data. This also leads to lower test results of the model. Experimenting with layer sizes shows out that layer sizes between 300 and 550 work best for this task. In addition, the sequent layers should decrease in their size from round about 500 to 300 to reach the best results (experiments 159-184).

The third applied strategy is dropout, a standard tool for reducing overfitting and enforcing a better generalization by deactivating nodes during training [21]. But enforces the model to build up more redundancies and lowers the amount of information for each layer to enhance the ability to generalize the data. But experiments 38 and 49-51, shown in figure 4.3, state that using dropout is not helpful to increase accuracy. It is counterproductive because instead of increasing the level of generalization and therefore improving the accuracy, the accuracy drops both on the train dataset and on the test dataset. Using a dropout value of 0.2 reduces the accuracy to just 61%. Increasing the level of dropout lowers the accuracy more and more until at a dropout level of 0.4 the whole model performs nearly similar than random choosing and reaches an accuracy of only 1.2%. Therefore, dropout is in this stage not helpful but will be discussed later again for learning on datasets with reduced amounts of measurements.

A further strategy is using larger datasets for training. Using larger datasets enables the usage of larger models. Therefore, using more data is itself not a strategy to enhance the results, except one case. Using larger datasets can increase

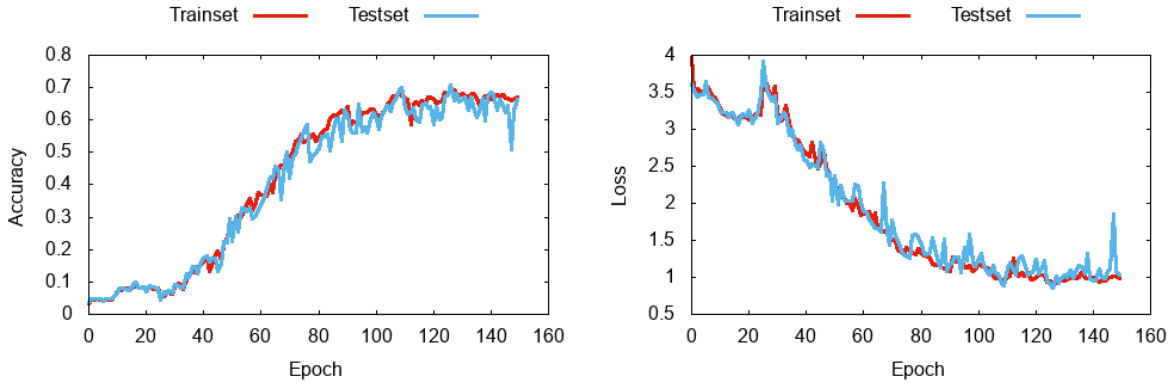


(a) Accuracy on train dataset.

(b) Accuracy on test set.

Figure 4.3: Comparing impact of different dropout levels on the train and test set

the accuracy of a model if a phenomenon called overfitting occurs in a model. Overfitting occurs often with large models and comparably small datasets. Using large data is one option to avoid this effect, another one - Dropout - was already discussed beforehand. Overfitting can be detected by comparing the accuracy on the training dataset and on a test set. If they differ significantly or differ more and more with time is a strong indicator that overfitting has occurred. But taking a look at the training process of the largest implemented model - the 20 layer model - shows no significant deviation (figure 4.4), which means no overfitting is occurring and the training dataset with 100.000 functions is large enough even for this huge model. Further data will, therefore, take only little effect, because the data the models are running on is stratified and contains in a set with term contribution 0, 800 functions per class. This seems to be sufficient.



(a) Comparing by the accuracy.

(b) Comparing by the loss.

Figure 4.4: Comparing the learning curves during training on the train set and the test set based on the accuracy after each epoch (a) and the loss (categorical cross entropy) after each epoch (b). The trained model is the basic 20-layer model with 200 nodes per layer. In both metrics, the accuracy, and the loss, no significant deviation is observable between evaluating on trainset and on test set, this shows that no overfitting occurs.

Another common strategy to enlarge the level of generalization is to use kernel regularization. Kernel regularization aims to restrict the kernel size to learn a simpler solution. This is done by adding a regularization term to the optimizer function. Applying this method prevents, in this case, the model from learning anything (figure 4.5). The maximum reached accuracy was 4.19% with an l2 regularization (0.01). Using l1 regularization or a combination of both regularizations ends up in 1.2% accuracy. Applying regularization is obviously not a good idea, because the same model without regularization reaches 70.76 % (experiment 234).

The sixth applied strategy is finetuning. For the core training process, the optimizers are adjusted to learn fast and increase the model's accuracy in a few epochs. Therefore, the learning rates of the optimizers are chosen higher, but this leads us to a less optimal solution. This means further refinements with a lower learning rate are possible. For that purpose, a second stage of training is applied, the finetuning, usually using a lower learning rate or a different optimizer. During this second training, the adjustable parameters in the model are only slightly updated to reach the local optimum.

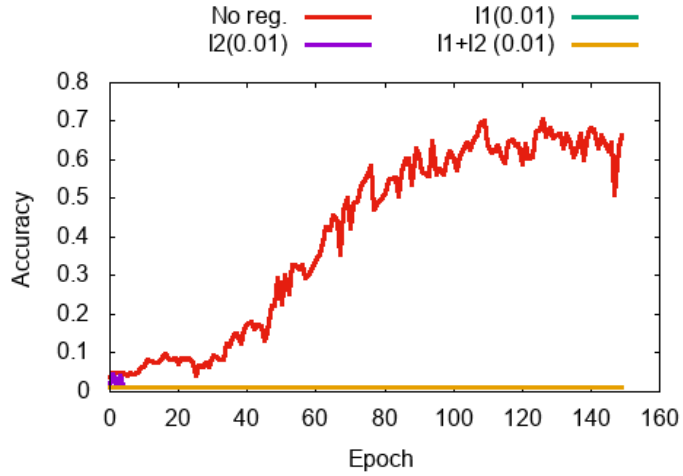


Figure 4.5: Comparing different regularizations with using no regularization. Any of these three regularization options (l1, l2 and l1+l2) results in an accuracy of less than 5%. Instead the baseline reaches without any form of kernel regularization an accuracy of 70.76%.

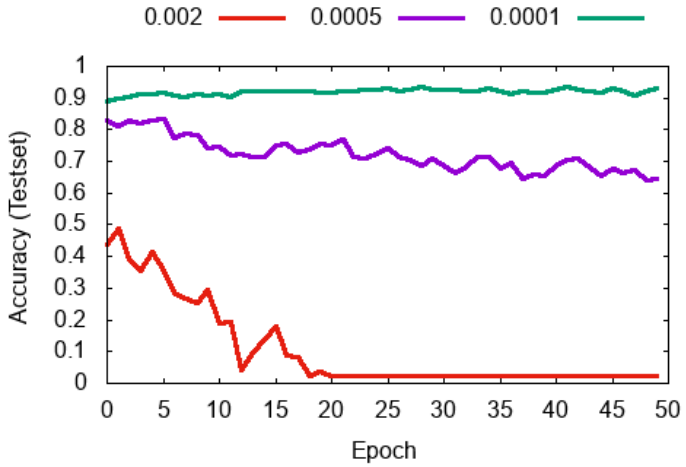


Figure 4.6: Evaluating different learning rates for the Nadam optimizer on finetuning. The keys denote the learning rate.

Finetuning cannot lead to a globally optimal solution, but it can refine the previously configured weights. Evaluation of this finetuning approach is done for multiple different optimizers with different learning rates, on an adjusted baseline reaching 89,78% accuracy on the test set (experiment 74). Table 4.2 shows out that often the assumed refinement lowers the accuracy significantly for some optimizers because these were not fitted to the task (already further discussed in section 4.2.1). Also, it confirms the intention of finetuning: the refined models are mostly performing better than the original one. Also tuning with a lower learning rate increases the accuracy to some point. Lowering the learning rate too much, gains after 50 epochs sometimes no better results, but for more epochs, it will. Based on these results, using Adagrad with a learning rate of 0.0005 or Adam with a learning rate of 0.0001 is the best option for finetuning.

Besides choosing the optimizer, a good learning rate is very crucial for finetuning. Figure 4.6 shows the learning rates of three finetuning experiments using all Nadam as the optimizer. The influence of the learning rate is significant. Using a default learning rate (default in the tf.keras implementation) lower the accuracy until the classes were nearly randomly chosen. But with lower learning rates, this optimizer can be successfully used for finetuning. This means that the observations about the optimizers for the core training process were not applying to finetuning. Finetuning with optimizers like Adam, Adagrad and Nadam are outperforming the baseline, if the learning rate is well chosen.

Applying finetuning (Adagrad, lr=0.0005) to the trivial model with only one hidden layer, improves the accuracy from 51.2% to 62.9%. These differences illustrate the influence of the learning rate to the final result quality very well.

FT Optimizer	Learning Rate	Accuracy after 50 epochs
Adagrad (default)	0.01	0.8042
Adagrad	0.0005	0.9405
Adagrad	0.0001	0.9269
Adam (default)	0.001	0.7478
Adam	0.0005	0.8459
Adam	0.0001	0.9463
RMSprop (default)	0.001	0.8093
RMSprop	0.0005	0.8769
RMSprop	0.0001	0.9333
Nadam (default)	0.002	0.4891
Nadam	0.0005	0.829
Nadam	0.0001	0.9359
SGD (default)	0.01	0.5016
SGD	0.005	0.4452
SGD	0.001	0.3088
Adadelta (default)	0.001	0.902
Adadelta	0.0005	0.9312
Adadelta	0.0001	0.9201

Table 4.2: Accuracy for different finetuning configurations. The configurations vary in the used optimizer and the chosen learning rate. The baseline model, where the finetuning's are compared on, has an accuracy of 89,78%. Listed configurations are the experiments 74 to 93, excluding experiment 88.

Configuration	Chosen Option
Batch size	300
Epochs	150
Loss Function	Categorical Crossentropy
Optimizer	Adadelta with learningrate 0.001
Finetuning Optimizer	Adagrad with learningrate 0.0005
Finetuning Epochs	150

Table 4.3: Configuration of the found best performing model.

4.2.3 Optimal Configuration

Putting all the strategies, which were successfully raising accuracy, leads to a model with 98.44% accuracy on the usual 10k test set. It contains 9 hidden layers shown in figure 4.7, each one using $tanh$ as activation function. The layer dimensions are in general decreasing, except hidden layer 4 and 5. It shows out that choosing these two layers with a higher dimension increases the model's accuracy. Besides these two layers, the number of nodes is decreasing. The last three layers of the model share the same dimension, they are containing 350 nodes. The nodes of each layer are flat aligned, and the layers are densely connected. The used optimizer is Adadelta with a learning rate of 0.001. The batch size is chosen to 300 and the training duration to 150 epochs (table 4.3). The model is trained on the 150k dataset with term contribution 6% and noise 2%. After training the model first time, finetuning with optimizer Adagrad and a learning rate of 0.0005 is applied. The finetuning lasts for 150 epochs again. Training takes at all 27 minutes, 12 minutes for finetuning and 15 for the first training on Nvidia Geforce GTX 1080 Ti. The averaged f1-Score of the single classes is 0.98 and therefore pretty well. More details about the f1-Score of the single classes can be found appended in section 6.2. It shows out that this model performs well for nearly all classes. The f1-score of each class is in minimum 0.87 even for classes with low support, except for class 87. Class 87 just has a support of 2 and its both datasets were classified falsely. In contrast, class 28 also shows a support of 2 and its both datasets are successfully correct classified. This means the model is partially able to classify under-represented data correctly. The f1 scores and support of each class are visualized in figure 4.8.

Figure 4.9 shows the accuracy improvement during the whole training process. The finetuning raises the accuracy to more than 98% while without only 91.6% accuracy was reached on the test set.

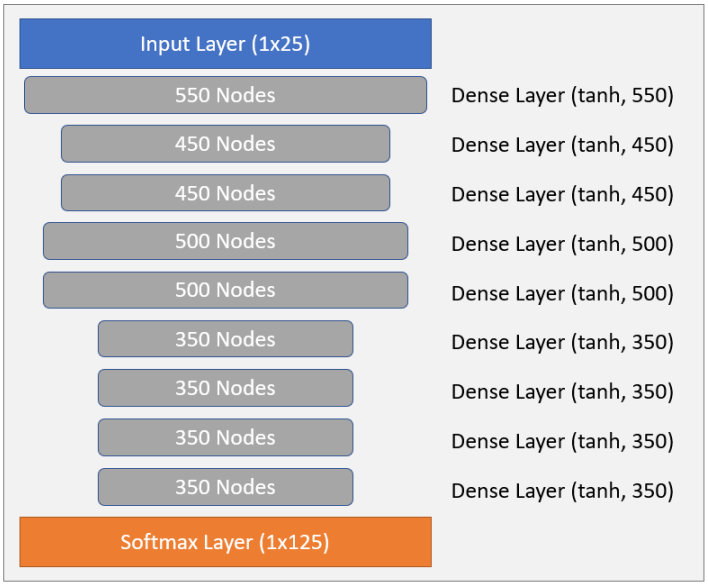


Figure 4.7: Layer overview of the best performing model (experiment 184.2). It reaches an accuracy of 98,44%. It contains 9 densely connected hidden layers with varying sizes and a following softmax activation layer as described in the figure.

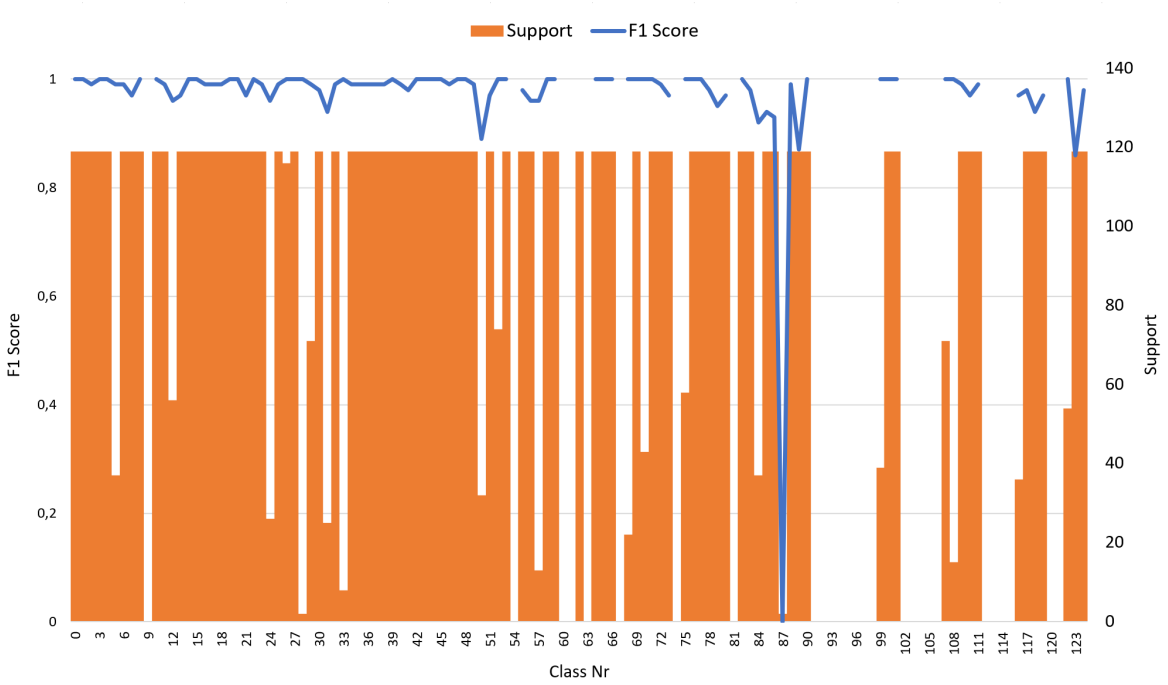
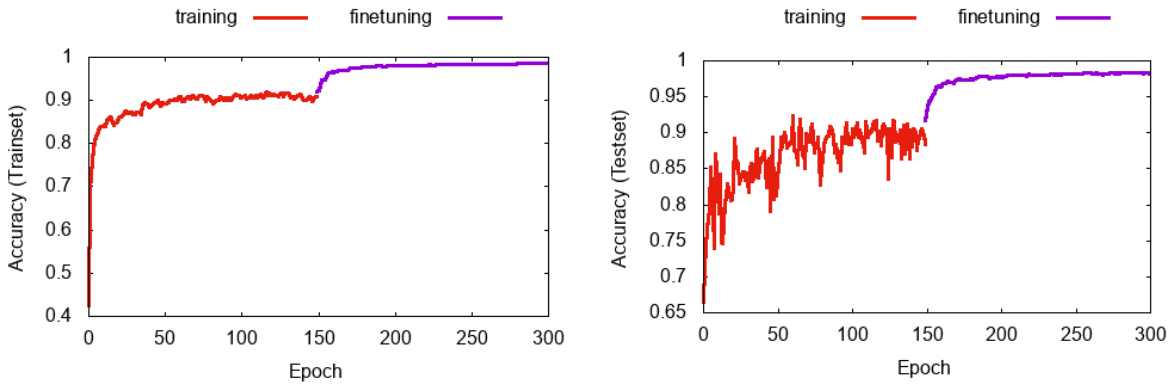


Figure 4.8: Diagram showing the support and f1 score of each of the 125 classes. On the horizontal axis, the classes are noted by their number (definition of class numbers and the corresponding function is appended in section6.1). The f1-score is denoted on the left vertical axis while the support is on the right vertical axis. Classes that are not covered in the dataset, because they are not generatable with a term contribution of 6%, got no value assigned.



(a) Accuracy on train set during training.

(b) Accuracy on test set during training.

Figure 4.9: Visualizing the accuracy during the training process (experiment 148-2), showing both training (tr) and finetuning (ft). Figure (a) shows the accuracy on the trainset and (b) on the test set. The x-axis denotes the epoch and the y-axis the actual accuracy after this epoch. The epochs are starting to count with 0. There are small gaps between the training and the finetuning line because before finetuning the best model during training was loaded. This was done by observing the accuracy on an evaluation set after each epoch during training with a checkpointer.

4.3 Measurement reduction

Beforehand the model was trained on datasets with 25 measurements each. Executing this amount of measurements is very costly. Therefore, for a beneficial usage of this model, a decrease in the measurements is important, also because other non-machine-learning based models come along with only 10 to 12 measurements. Thereby the goal of this new model is to come along with less than half of the measurements, measurable with few costs in addition.
For using fewer measurements there are two possibilities of how to set up and train the model. The first option is to train a separate model for each amount of measurements. This means for each amount of measurements a single model, separately trained. This is furtherly called a single-measurement-amount-model. Besides that, the other option is to train one model, that is able to work with varying amounts of measurements. This is done by keeping the input on the original size of 25 measurements but masking the positions out where no measurement was taken, by setting these measurements to zero. This will further be called the multi-measurement-amount-model, a single model that is able to solve tasks with different amount of measurements.

Besides, both methods use the same reduction strategy, which defines for each amount of measurements there they should be taken. The defined order for which amount of measurements which configurations are used is shown in 4.10. Example for a data layer with 9, 11 and 15 measurements are shown in figure 4.11. The general schedule for taking measurements is to measure first the configurations with minimal x and y. Afterwards, the configurations between the two axes, starting from the bottom left corner to the top right.

The idea behind this prioritization of measurement configurations with minimal x and y values is on the one hand to minimize the measurement costs. If for example, x is denoting the problem size, measuring for a smaller problem size is cheaper than for larger sizes. On the other hand, information about runtime with high chosen x and y should be kept. Therefore for 9 measurements, it is required to input the 5 measurements taken on $y = 4$ and the remaining measurements taken for $x = 10$. More measurements will then start to fill the grid from the left bottom corner of the grid. This reduces the measurement effort while keeping as much information as possible. If there are less than 9 measurements, there will be scattered equally on the minimal configurations for $x = 10$ and $y = 4$.

Single-measurement-amount learning and multi-measurement-amount learning are applied on different data. Each single-measurement-amount-model uses for training only data with the specific amount of measurements, chosen accordingly to the previously described order. They were all using the previously found optimal model, but with masked out measurements. This means the input contains still 25 values, but the coordinates were no measurements were taken are set to 0. This enables the opportunity to reuse the previously found optimal model with its 98% accuracy and compare the results of the single-measurement-amount and multi-measurement-amount learner. The data for training is still the 100k dataset having a term contribution of 6% and a noise of 2%. Therefore, for each single-measurement-amount-learner, the surplus measurements are masked out. In difference the multi-measurement-amount learner is

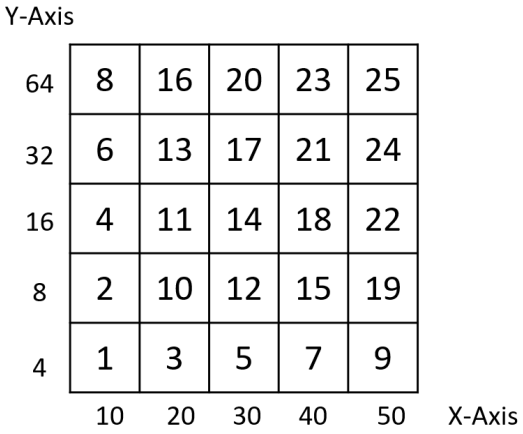
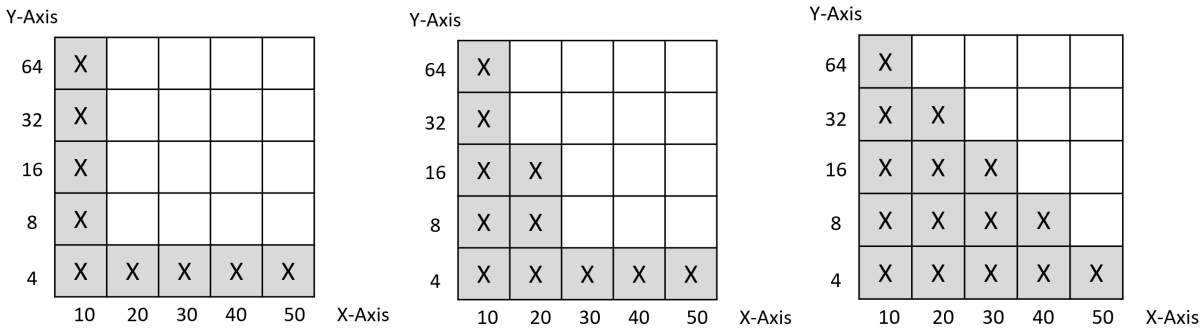


Figure 4.10: Diagram showing the priority between the measurement configurations. 1 denotes the highest priority, while 25 the lower. First the measurement for x minimal and y minimal are taken, afterwards the configurations near to the bottom left corner of the grid.



(a) 9 measurements. **(b)** 11 measurements. **(c)** 15 measurements.

Figure 4.11: Chosen configurations for 9 measurements (a), 11 measurements (b) and 15 measurements (c). Each grid element represents one measurement configuration. A X inside means that a measurement is taken for this configuration. An empty field means that not measurement is taken and therefore this field is masked out and set to zero or removed from the dataset, depending on the approach.

trained still on the 100k_6-2 dataset but with different amounts of measurements. Therefore, the dataset is split, and each part is masked to the corresponding amount of measurements so that finally the amounts of unmasked measurements are equally distributed over the whole dataset. In addition, the dataset is randomly permuted before training was applied. These two methods are both applied on measurement sizes between 9 and 25. Comparing the trained models, the 17 single-measurement-amount-models and the multi-measurement-amount-model, it shows out that the single-measurement-amount-models are performing better than the multi-measurement-amount-model (figure 4.12). It shows out that the multi-measurement amount model does not reach more than 95% accuracy and performs for each amount of measurements chosen from the interval of 9 to 25 inferior to the single-measurement-amount-model for this amount of measurements.

In addition, it shows out that using more datapoints for training does not improve the results infinitely often. Instead, the maximal accuracy of 99.2% is reached by a single-amount-measurement model with 23 measurements. The same behavior can be observed on the multi-measurement-amount-model. The maximum accuracy of 94.7% is reached on 22 and 23 measurements. This behavior was already previously found by the work of Marcus Ritter and is hereby confirmed. In general, both methods reach for only 9 measurements still an accuracy of more than 60%, the single-measurement-amount model for 9 measurements even 73%. Additionally, both methods show a drop in accuracy with less than 15 measurements. Choosing a lower amount of measurements decreases the accuracy strongly while using a higher amount of measurement makes only less difference. Therefore, it is recommended to use the single-measurement-amount-model with 15 measurements if possible. Choosing also 15 measurements for the multi-measurement-amount model might make sense but if higher accuracy is required, using 19 measurements is advised. Using 19 measurement instead of 15 increases the accuracy from 88% to 94%.

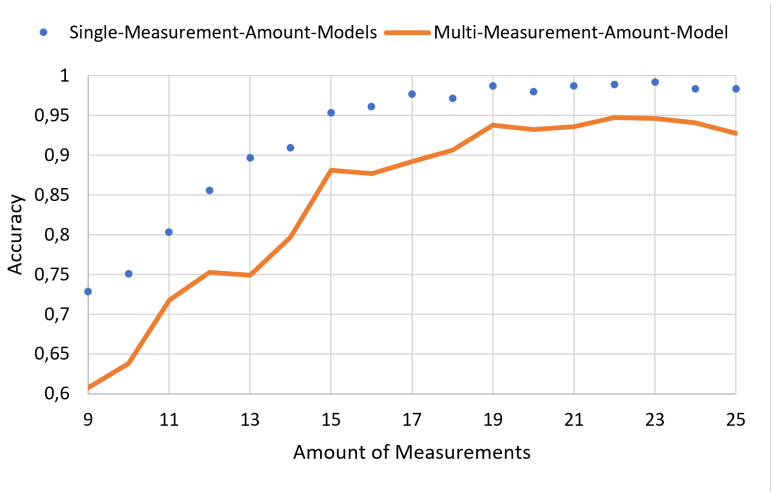


Figure 4.12: Chart showing the accuracy on the y axis for each amount of measurements denoted on the x axis for the corresponding single-measurement-amount-model and the multi-measurement-amount-model.

4.4 3 parameter approaches

4.4.1 Learning configurations (optimizer, batch size)

Configuring the learning parameters of the 3 parameter approaches can be done by taking the results of the exhaustive parameter search of the two-parameter models into account. This is possible, due to the similarity of the underlying problems: The three-parameter problem is solving, in general, the same task, just with a further dimension. But the task still diverges from the 2-parameter task in terms of the number of measurements and the additional dimension. Therefore, the results of the two-parameter model are applied and just the possibly diverging parameters (batch size and optimizer) are evaluated again. The batch size must be evaluated again because choosing an optimal batch size heavily depends on the label space and the training data size. The label space is enlarging significantly from 125 to 4625, which requires to evaluate the batch size again. In addition, the training data size is increased to $1 * 10^6$ functions, which will slow down the training process, if the batch size is still in the range of the two-parameter model batch sizes. Besides, the slowdown of the training process will be intensified by probably using larger networks, with more trainable parameters. Evaluating batch sizes between 1000 and 3000 on a basic implementation shows no significant differences between the batch sizes. Running the baseline model with a batch size of 1000 results in an accuracy of 0,75134 while running with batch size 3000 results in a similar even slightly better accuracy of 0,75745. Based on this observation the batch size is chosen based on their influence on the performance. It shows out that the configuration with batch size 1000 takes 1.6 minutes for training one epoch while the batch size 3000 model just takes 2 minutes on a Nvidia Geforce GTX 1080 Ti. Based on these observations, batch size 3000 is chosen for the 3 parameter approach.

Searching the best fitting optimizer for the two-parameter model shows that Adadelta and Adam are the best options for the core training process and Adagrad and Adam for the finetuning process. It shows out that the decision using Adadelta for training and Adagrad for finetuning is still the best option as shown in figure 4.13 and figure 4.14. Using Adam as optimizer instead would lower the accuracy from 32% with Adadelta to 11% with Adam. Also, for finetuning, a replacement of the optimizer chosen for the two-parameter models would lower the accuracy from 86.3% with Adagrad to just 70.6% with Adam as finetuning optimizer. This shows that the optimizers chosen for the two-parameter model are still the best option for the three-parameter model.

Evaluating the number of layers shows that models with less but larger layers reach a higher accuracy than adding more layers. This shows out by comparing a 10 hidden layer model with layer sizes varying between 800 and 1000, a 6 hidden layer model with layer node sizes between 800 and 1000 too and a 2 hidden layer model with 2000 nodes per layer. The training results are increasing with fewer layers from 75.74% on the 10 hidden layer model and 76.71% on the 6 hidden layer model to 82.55% on the 2 hidden layer model. This result is indicating that broader layers have a higher influence on the results than more layers.

Applying an evaluation of the ideal amount of layers on the optimal layer sizes of 4000 shows that there is still an increase in accuracy on adding further layers. Using a model with three hidden layers, each containing 4000 nodes, instead of a model with two hidden layers, each containing 4000 nodes too, outperforms the accuracy from 86.32% (experiment

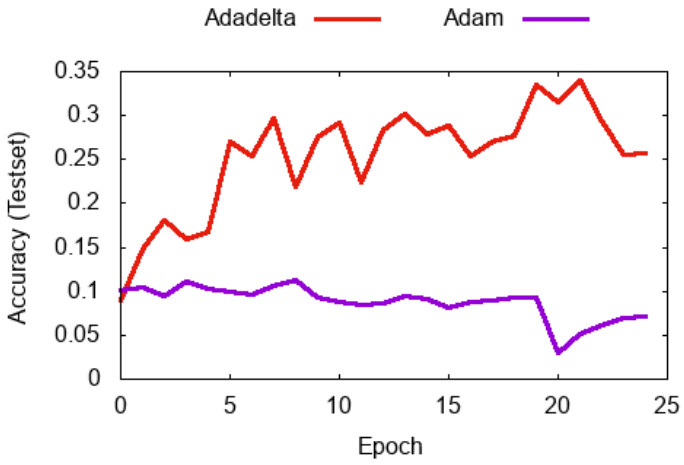


Figure 4.13: Comparing the optimizers Adadelta and Adam for the core training process. The accuracy is denoted on they y-axis and the corresponding epoch on the x-axis.

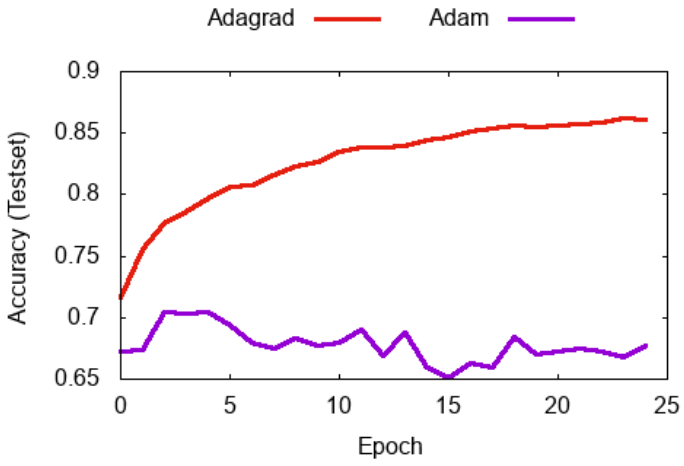


Figure 4.14: Comparing the optimizers Adagrad and Adam for the finetuning process. The accuracy is denoted on they y-axis and the corresponding epoch on the x-axis.

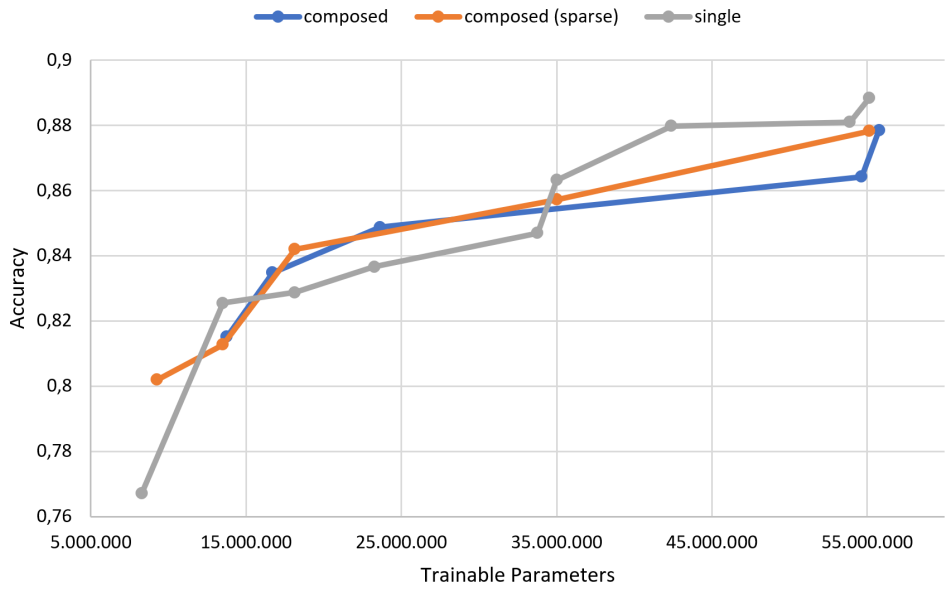


Figure 4.15: Comparison of the 3 parameter approaches, single-3-parameter-model, composed-3-parameter-model, composed-3-parameter-model(sparse), based on accuracy denoted on the y-axis and number of trainable parameters on the x-axis. Visualization uses the pareto-optimal results of all experiments of each approach, chosen by the experiments amount of trainable parameter and its achieved accuracy.

248) to 87.79% (experiment 256) by 1.47 percentage points, but the amount of trainable parameters and therefore also the training time raises largely. The model with two hidden layers contains 35 million trainable parameters, while the three hidden layer model contains 51 million. This improvement of nearly 1.5 percentage points comes along with an increase of 45% of the trainable parameters. This shows that the effort to reach a better accuracy is from the point of models with two hidden layers pretty costly because there is less increase in accuracy but a major increase in the number of trainable parameters and memory consumption and also time for training as well as for testing. In these two experiments, the training times are pretty similar with 2 hours while the consumed memory raised from roughly 38 GB to 58 GB.

Due to these reasons experimenting with further layers had not been performed, but on large clusters with a hundred gigabytes of RAM in minimum, adding further layers to the model will probably achieve better results while consuming enormous more memory and computing power.

4.4.2 Approach Comparison

Section3.3 introduces two general types of approaches: the 3-parameter-approach and the composed-3-parameter-approach. Further, the composed-3-parameter-approach can be implemented using the whole class-probability distribution or the sparse argmax value. The idea behind the composed-3-parameter-approach was to take the results of a pretrained two-parameter model on a reduction of the dataset into account to predict better results with a smaller model. During the varying experiments with the aim to optimize the three approaches it shows out that for an intermediate area of trainable-parameters of the models, the aim of reaching better results with the composed-approach was reached. Figure 4.15 shows that for models with an amount of trainable parameters between 16 and 34 million, the composed approaches are performing better than the single-three-parameter approach. With a larger amount of trainable parameters, the single approaches are performing better.

4.4.3 Optimal Model Evaluation

Putting the previously evaluated strategies all together results in an optimal model that reaches an accuracy of 88.84% and in addition a weighted averaged f1 score of 0.88. This model contains three hidden layers, followed by a softmax activation layer. A model overview is shown in figure 4.16. The first hidden layer contains 5000 nodes and the following two ones both 4000. The amount of trainable parameters adds up to 55 million and is by that pretty near the largest implemented model. The optimizer for the core training process is Adadelta with a learning rate of 0.001 and for

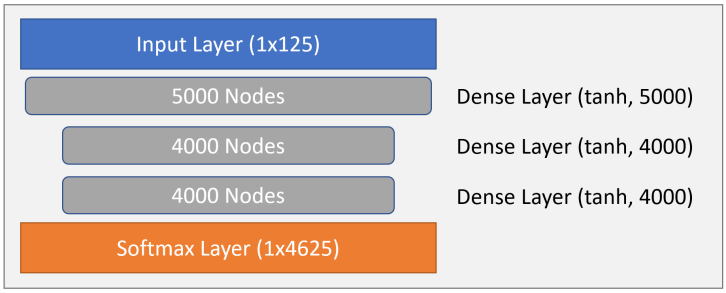


Figure 4.16: Layer overview of the best performing model (experiment 261). It reaches an accuracy of 88.84%. It contains 3 densely connected hidden layers with varying sizes and a following softmax activation layer as described in the figure.

finetuning Adagrad with a learning rate of 0.0005 is used. The training is performed with a batch size of 3000 on the 1m dataset coming along with 6% minimum term-contribution and 2% noise.

4.4.4 Measurements Reduction

Equally to the two-parameter approach, a reduction of measurements here is beneficial too. Requiring 125 measurements would be very costly and time-consuming. Therefore a reduction of the parameters is essential for a real-world application. To achieve the goal of expecting fewer measurements and only less resource-consuming ones the same two options as in the two-parameter approach are evaluated. One option is to have a certain model for each amount of measurements. This means that beforehand 113 models need to be trained for supporting numbers of measurements between 13 and 125. Each model is then applicable to just data with the same amount of measurements. The fitting model from these bunch of models can be chosen by summing up the given measurements and choosing the corresponding model. Besides this advantage of having a model that is fully trained on the specific amount of measurements, the training and storing of all these models is a big drawback. In contrast to these single-measurement-amount-models, it is also possible to train one model on data with varying amounts of measurements similar to the two-parameter model. For training this multi-measurement-amount-model the data will get compounded out of dataset shrank to all the possible numbers of measurements. Discarded measurements are masked out to still have the same dataset sizes, but the model is thereby forced to focus on just the measurements who were not masked out. The measurement reduction operation is in concept similar to the one applied for two parameters but is dealing now with three parameters. The main idea is still to consider the measurement configurations with the lowest effort required for measuring them. For the decision which measurement configurations to take for which amount of measurements the configurations are mapped to a $5x5x5$ grid, where each grid-cell denotes one measurement configuration. Now, this grid is shifted so that the grid cell containing the minimal configuration of all three parameters is at the origin of the coordinate space. Also, the grid will be aligned to the coordinate axes so that three grid borders are matching coordinate space axes while holding the condition that the minimal parameter configuration grid cell is still at the origin. Then the decision about the consideration of the grid cells can be done. The values on the axes are chosen first. Afterwards, the grid cells are ranked by their manhattan distance to the origin, calculated on the three coordinate space axes.

Training the multi-measurement-amount model and the single-measurement-amount-models reach results shown in figure 4.17. Similar to the two-parameter scenario the single-measurement-amount-models are performing obviously better than multi-measurement-amount-model. The single-measurement-amount-models are performing pretty constant 14 percentage points better in average. They are reaching a maximum accuracy of 88.5% while the multi-measurement-amount-models just reach 74.5% both with 125 measurements. It shows out that considering further measurements has a higher impact on models with less than 37 measurements than for models with higher amounts. The accuracy is increasing constantly with each further measurement roughly until 37 measurements are reached. Afterwards, there is still a constant improvement, but lower than at the beginning. This behavior can be observed for the single-measurement-amount-models as well as for the multi-measurement-amount-model. At this changing point at 37 measurements, the accuracy of the single-measurement-amount-model is still at 74% and 64% for the multi-measurement-amount-model. Evaluating the approach on just 13 measurements, the minimal configuration with just the measurements on the three axes, shows that with the multi-measurement-amount-approach an accuracy of almost 50% has been reached while the single-measurement-amount-model reaches outstanding 65.8%. This evaluation of the results with fewer measurements shows that the model's accuracy is decreasing with fewer measurements slowly until 37 is reached, afterwards the

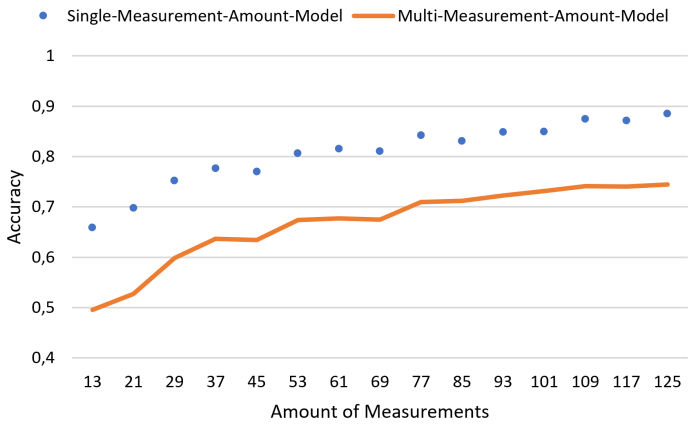


Figure 4.17: Chart showing the accuracy of the single-measurement-amount-models (one model per amount of measurements) and the multi-measurement-amount-models (one model for dealing with varying numbers of measurements) on the y-axis for amount of measurements denoted on the x-axis.

decrease is strengthened. Therefore it is advised to use in minimum 37 measurements which results in accuracies of more than 64%/74%.

4.5 Comparing with State of the art results

In 2017 Reisert et al.[10] proposed a state-of-the-art approach for finding performance functions. This approach is based on the Parallel Model Normal Form (PMNF) too and was evaluated on functions with common terms $(1, x, x^2, x^3, log_2(x))$ and is therefore comparable to the deep-learning-based model introduced by this work. Their model evaluation is based on 1000 randomly generated functions and done with two metrics:

- Lead Order Term Matching
- Prediction within ±2% noise

In a 2-parameter environment (functions based on PMNF with n=2) they reached an accordance of the lead order term on roughly 86% of the tested functions. The deep learning two-parameter model was able to successfully predict the whole complexity function in more than 98% of 100 thousand randomly generated functions. The deep learning model is coming along with an improvement of more than 10 percentage points comparing their lead order term accordance with the Deep Learning Models full function accuracy. This shows out that the deep learning model introduced by this work can compete successfully with state-of-the-art approaches and outperforms them both in the percentage of accordance and exactitude of the predicted function.

5 Conclusion

Neural networks can predict the complexity class of the performance function out of several measurements very well. The used deep learning model, named as multi-layer-perceptron or feed-forward-neural-network, are successfully applied. Scenarios with 2 performance relevant parameters can be solved with an accuracy of 95% and above if 15 and more measurements are given to the model. Accuracy of up to 99% can be reached if 23 and more measurements are available as shown in chapter 4.3 and therefore the approach can be evaluated as solving the problem perfectly. Further, both the accuracy and the f1 score of the model are reaching 99% in maximum. This model is beneath classes with a high support also able to classify classes with a comparably small support very well. Most of the classes with 10 times less support than average are correctly classified. As shown in section 4.5 the deep learning model outperforms current state-of-the-art approaches like a novel ExtraP algorithm introduced in Following The Blind Seer significantly both in accuracy and in granularity of the results too. The improvement in accuracy adds to more than 10 percentage points determined by the classification of the whole complexity function in the deep learning model compared to predicting just the lead order term in the state-of-the-art approach. This shows that using deep learning is, in general, a good method to get a prediction of which complexity class a performance function has. Afterwards, classic heuristic approaches like used in ExtraP are then able to predict a more accurate and faster prediction about the detailed performance function. Due to its efficiency and success metrics, this approach based on neural networks might improve and speedup current algorithms very well. Scenarios with more than 2 performance relevant parameters are actually way more difficult to solve with heuristic approaches but it shows out that the same applies also on neural networks. The large label space and the increase of the measurements is a challenge also for MLPs, due to its outcome of neural networks with significant more trainable parameters, which are slowing down the training process, and the large label space. Besides the challenges, the model is still able to perform very well and reach accuracies of 88.8% and a weighted f1 score of 0.88 in maximum for three parameters. Using a composed model of a two-parameter-model which supports a three-parameter-model shows up advantages for middle-sized three-parameter models. Models with an amount of trainable parameters in the range between 16 and 34 million parameters can be improved by 1 percentage point in average by using a two-parameter model for extracting further features and padding them to the input data. Besides this range of trainable parameters, models with a higher amount, perform better without this additional information. The reason behind might be that models with more parameters are able to handle the whole problem pretty well on their own and further data might just shift the focus away from the original data to the new, but less relevant information, in case of the ability to handle the problem very well by themselves. Also having in addition to the 3 parameter measurements, a prediction about the complexity class of the first 25 parameters in a 2 parameter space might help just to a certain point, because the task for finding the complexity function including all three parameters still needs to be done. As long as a large single three-parameter-model is not solving the task perfectly, a composed model cannot reach perfect results too. This shows out to be a difficulty for predicting higher n-parameter models with the evaluated deep learning approaches. Besides that, the result of using deep learning for predicting a complexity class shows that this work might be a good baseline for future research in using deep learning to support empiric performance modelling approaches.

5.1 Limitations of the approach

The two-parameter approach can be applied very well for largely varying performance functions. But the output of the model, the complexity class, is limited to the basic complexity function building blocks. Performance functions containing $log(x) * x$ or $log(x)^2$ are currently not predictable by the model. The range of covered function is broader than in ExtraP but the granularity of the output is still limited to some often occurring function terms.

A further limitation is that terms with a small impact cannot be detected with this model. This model is trained on data with a minimum term contribution of 6%, therefore functions like $x^2 + log(y)$ (contribution of log-term at minimum configuration is 2% and at maximum configuration 0.2%) cannot be detected, even if the level of noise is low enough to theoretically label it correct.

In addition, a minimum of 15 measurements is required. Having less than 15 available measurements drops the performance of the model significantly. The results are still usable but were veering away from an accuracy of 90% and above.

Expecting measurements taken on specific predefined performance relevant parameter configurations limit the applicability of the approach to data were these measurements already exists or these measurements can be taken afterwards.

The model is not able to deal with measurements taken on arbitrary configurations.

In addition, the three-parameter model has some more critical limitations. The level of accuracy is not that well than for 2 parameters. Only a level of 88.8% accuracy can be reached in maximum on taking all the 125 measurements into account. For real-world applications, this amount of measurements might be not viable and might prevent the approach from using it. Also, in the three-parameter approach, the range of covered functions is way more restricted compared with the theoretical function space including three parameters. The predicted function may contain at most two added terms and no further. This limitation is crucial for lowering the label space but also makes the outcome for functions with more terms or different complexity function building blocks inaccurate.

5.2 Outlook and Future Work

This work shows that in general, the problem is very well solvable with neural networks. This leads to multiple works that can be done in the future. Further work is to extend the two and three-parameter approaches to use arbitrary measurement coordinates instead of predefined configurations. That would enable to use all the taken measurements and not only ones fitting to the predefinition. This might be possible in the future by adding further information to each measurement. Depending on the concrete implementation and level of independence from the configurations, different deep learning methods might be helpful. Using an encoder-decoder-model with a recurrent neural network as proposed by Cho et al. [22] might allow dealing with varying amounts of measurements and also prevents the size of the model from exploding, when each input gets in addition to the measurement the configuration information. After inserting all the measurements, one configuration and its measurement in each timestep, the RNN in the encoder has developed an internal representation about the data, something similar to embeddings, and a decoder can predict the complexity class afterwards on this representation.

Further, it might be worth it to elaborate whether it's possible to apply this approach on finding complexity changing points in the performance of a program [23].

Further work in the field of the three-parameter and in future also n-parameter approaches would be to enhance the strategy of reusing n-1 parameter models for solving an n-parameter task. This would enable the model to take further information into account and therefore the size of the model and the number of trainable parameters can be downsized. In addition, also an optimization of the proposed MLP approach is possible.

Also, with an increasing number of parameters, the models are more and more struggling with the label space. Taking all possible complexity functions into account increases the label space largely. Therefore, a method to reduce the number of complexity classes while not making the outcomes too inaccurate and still keep a proven granularity would be beneficial.

The reached accuracy showed that the model works very well. Therefore, it would be interesting to include these models in ExtraP and evaluate how the overall performance of ExtaP algorithms change.

In general, implementing the proposed approaches in ExtraP will make comparisons between the heuristic approaches and these deep learning approaches possible. Based on these results, further knowledge about using deep learning models for enhancing existing performance modelers can be gained and statements about the level of improvement can be made.

6 Appendix

6.1 Complexity Functions List (2 Parameter Model)

Table 6.1: Table with all 2 parameter classes and their associated class number.

class nr	function
0	1
1	$log(\text{size}, 2)$
2	size
3	$\text{size} + log(\text{p}, 2)$
4	size^2
5	$\text{size}^2 + log(\text{p}, 2)$
6	$\text{size}^2 + log(\text{p}, 2) * log(\text{size}, 2)$
7	$\text{size}^2 + \text{p}$
8	size^3
9	$\text{size}^3 + log(\text{p}, 2)$
10	$\text{size}^3 + log(\text{p}, 2) * log(\text{size}, 2)$
11	$\text{size}^3 + log(\text{p}, 2) * \text{size}$
12	$\text{size}^3 + \text{p}$
13	$\text{size}^3 + \text{p} * log(\text{size}, 2)$
14	$\text{size}^3 + \text{p}^2$
15	$log(\text{p}, 2)$
16	$log(\text{p}, 2) + log(\text{size}, 2)$
17	$log(\text{p}, 2) * log(\text{size}, 2)$
18	$log(\text{p}, 2) * log(\text{size}, 2) + \text{size}$
19	$log(\text{p}, 2) * \text{size}$
20	$log(\text{p}, 2) * \text{size} + \text{size}^2$
21	$log(\text{p}, 2) * \text{size} + \text{p}$
22	$log(\text{p}, 2) * \text{size}^2$
23	$log(\text{p}, 2) * \text{size}^2 + \text{size}^3$
24	$log(\text{p}, 2) * \text{size}^2 + \text{p}$
25	$log(\text{p}, 2) * \text{size}^2 + \text{p} * log(\text{size}, 2)$
26	$log(\text{p}, 2) * \text{size}^2 + \text{p}^2$
27	$log(\text{p}, 2) * \text{size}^3$
28	$log(\text{p}, 2) * \text{size}^3 + \text{p}$
29	$log(\text{p}, 2) * \text{size}^3 + \text{p} * log(\text{size}, 2)$
30	$log(\text{p}, 2) * \text{size}^3 + \text{p} * \text{size}$
31	$log(\text{p}, 2) * \text{size}^3 + \text{p}^2$
32	$log(\text{p}, 2) * \text{size}^3 + \text{p}^2 * log(\text{size}, 2)$
33	$log(\text{p}, 2) * \text{size}^3 + \text{p}^3$
34	p
35	$\text{p} + log(\text{size}, 2)$
36	$\text{p} + \text{size}$
37	$\text{p} + log(\text{p}, 2) * log(\text{size}, 2)$
38	$\text{p} * log(\text{size}, 2)$
39	$\text{p} * log(\text{size}, 2) + \text{size}$
40	$\text{p} * log(\text{size}, 2) + \text{size}^2$
41	$\text{p} * log(\text{size}, 2) + log(\text{p}, 2) * \text{size}$
42	$\text{p} * \text{size}$

43	$p * size + size^2$
44	$p * size + size^3$
45	$p * size + log(p, 2) * size^2$
46	$p * size + p^2$
47	$p * size^2$
48	$p * size^2 + size^3$
49	$p * size^2 + log(p, 2) * size^3$
50	$p * size^2 + p^2$
51	$p * size^2 + p^2 * log(size, 2)$
52	$p * size^2 + p^3$
53	$p * size^3$
54	$p * size^3 + p^2$
55	$p * size^3 + p^2 * log(size, 2)$
56	$p * size^3 + p^2 * size$
57	$p * size^3 + p^3$
58	$p * size^3 + p^3 * log(size, 2)$
59	p^2
60	$p^2 + log(size, 2)$
61	$p^2 + size$
62	$p^2 + size^2$
63	$p^2 + log(p, 2) * log(size, 2)$
64	$p^2 + log(p, 2) * size$
65	$p^2 + p * log(size, 2)$
66	$p^2 * log(size, 2)$
67	$p^2 * log(size, 2) + size$
68	$p^2 * log(size, 2) + size^2$
69	$p^2 * log(size, 2) + size^3$
70	$p^2 * log(size, 2) + log(p, 2) * size$
71	$p^2 * log(size, 2) + log(p, 2) * size^2$
72	$p^2 * log(size, 2) + p * size$
73	$p^2 * size$
74	$p^2 * size + size^2$
75	$p^2 * size + size^3$
76	$p^2 * size + log(p, 2) * size^2$
77	$p^2 * size + log(p, 2) * size^3$
78	$p^2 * size + p * size^2$
79	$p^2 * size + p^3$
80	$p^2 * size^2$
81	$p^2 * size^2 + size^3$
82	$p^2 * size^2 + log(p, 2) * size^3$
83	$p^2 * size^2 + p * size^3$
84	$p^2 * size^2 + p^3$
85	$p^2 * size^2 + p^3 * log(size, 2)$
86	$p^2 * size^3$
87	$p^2 * size^3 + p^3$
88	$p^2 * size^3 + p^3 * log(size, 2)$
89	$p^2 * size^3 + p^3 * size$
90	p^3
91	$p^3 + log(size, 2)$
92	$p^3 + size$
93	$p^3 + size^2$
94	$p^3 + size^3$
95	$p^3 + log(p, 2) * log(size, 2)$
96	$p^3 + log(p, 2) * size$

97	$p^3 + log(p, 2) * size^2$
98	$p^3 + p * log(size, 2)$
99	$p^3 + p * size$
100	$p^3 + p^2 * log(size, 2)$
101	$p^3 * log(size, 2)$
102	$p^3 * log(size, 2) + size$
103	$p^3 * log(size, 2) + size^2$
104	$p^3 * log(size, 2) + size^3$
105	$p^3 * log(size, 2) + log(p, 2) * size$
106	$p^3 * log(size, 2) + log(p, 2) * size^2$
107	$p^3 * log(size, 2) + log(p, 2) * size^3$
108	$p^3 * log(size, 2) + p * size$
109	$p^3 * log(size, 2) + p * size^2$
110	$p^3 * log(size, 2) + p^2 * size$
111	$p^3 * size$
112	$p^3 * size + size^2$
113	$p^3 * size + size^3$
114	$p^3 * size + log(p, 2) * size^2$
115	$p^3 * size + log(p, 2) * size^3$
116	$p^3 * size + p * size^2$
117	$p^3 * size + p * size^3$
118	$p^3 * size + p^2 * size^2$
119	$p^3 * size^2$
120	$p^3 * size^2 + size^3$
121	$p^3 * size^2 + log(p, 2) * size^3$
122	$p^3 * size^2 + p * size^3$
123	$p^3 * size^2 + p^2 * size^3$
124	$p^3 * size^3$

6.2 Class Evaluation Result List (2 Parameter Model)

Table 6.2: Evaluation results of the best found model (experiment nr. 184_2). For each class the precision, recall, f1-score and support are denoted. In addition the accuracy, the averaged f1-score and the weighted averaged f1-score (weighted accordingly to support).

class nr	precision	recall	f1-score	support
0	1.00	1.00	1.00	119
1	0.99	1.00	1.00	119
2	0.99	0.99	0.99	119
3	1.00	1.00	1.00	119
4	1.00	1.00	1.00	119
5	0.97	1.00	0.99	37
6	0.99	0.99	0.99	119
7	0.99	0.96	0.97	119
8	1.00	1.00	1.00	119
9	0.00	0.00	0.00	0
10	1.00	0.99	1.00	119
11	0.98	0.99	0.99	119
12	0.95	0.96	0.96	56
13	0.98	0.97	0.97	119
14	1.00	1.00	1.00	119
15	0.99	1.00	1.00	119
16	0.99	0.99	0.99	119

17	0.99	0.99	0.99	119
18	0.99	0.99	0.99	119
19	1.00	0.99	1.00	119
20	1.00	1.00	1.00	119
21	0.99	0.95	0.97	119
22	1.00	0.99	1.00	119
23	0.98	1.00	0.99	119
24	0.96	0.96	0.96	26
25	0.99	0.98	0.99	119
26	1.00	1.00	1.00	116
27	1.00	1.00	1.00	119
28	1.00	1.00	1.00	2
29	0.99	1.00	0.99	71
30	0.97	0.98	0.98	119
31	0.96	0.92	0.94	25
32	0.98	0.99	0.99	119
33	1.00	1.00	1.00	8
34	1.00	0.98	0.99	119
35	0.98	1.00	0.99	119
36	0.98	1.00	0.99	119
37	0.98	1.00	0.99	119
38	1.00	0.98	0.99	119
39	1.00	1.00	1.00	119
40	0.98	0.99	0.99	119
41	0.96	0.99	0.98	119
42	1.00	1.00	1.00	119
43	1.00	1.00	1.00	119
44	1.00	1.00	1.00	119
45	1.00	1.00	1.00	119
46	0.99	0.99	0.99	119
47	1.00	0.99	1.00	119
48	1.00	1.00	1.00	119
49	1.00	0.97	0.99	119
50	0.93	0.84	0.89	32
51	0.96	0.97	0.97	119
52	1.00	1.00	1.00	74
53	1.00	0.99	1.00	119
54	0.00	0.00	0.00	0
55	0.98	0.98	0.98	119
56	0.98	0.93	0.96	119
57	0.93	1.00	0.96	13
58	1.00	0.99	1.00	119
59	0.99	1.00	1.00	119
60	0.00	0.00	0.00	0
61	0.00	0.00	0.00	0
62	1.00	1.00	1.00	119
63	0.00	0.00	0.00	0
64	1.00	1.00	1.00	119
65	1.00	1.00	1.00	119
66	1.00	0.99	1.00	119
67	0.00	0.00	0.00	0
68	1.00	1.00	1.00	22
69	1.00	1.00	1.00	119
70	1.00	1.00	1.00	43
71	1.00	1.00	1.00	119

72	0.98	1.00	0.99	119
73	0.96	0.98	0.97	119
74	0.00	0.00	0.00	0
75	1.00	1.00	1.00	58
76	0.99	1.00	1.00	119
77	1.00	1.00	1.00	119
78	0.96	0.99	0.98	119
79	0.93	0.97	0.95	119
80	0.97	0.97	0.97	119
81	0.00	0.00	0.00	0
82	1.00	1.00	1.00	119
83	0.99	0.97	0.98	119
84	0.92	0.92	0.92	37
85	0.96	0.92	0.94	119
86	0.89	0.97	0.93	119
87	0.00	0.00	0.00	2
88	0.98	1.00	0.99	119
89	0.85	0.88	0.87	119
90	1.00	1.00	1.00	119
91	0.00	0.00	0.00	0
92	0.00	0.00	0.00	0
93	0.00	0.00	0.00	0
94	0.00	0.00	0.00	0
95	0.00	0.00	0.00	0
96	0.00	0.00	0.00	0
97	0.00	0.00	0.00	0
98	0.00	0.00	0.00	0
99	1.00	1.00	1.00	39
100	1.00	0.99	1.00	119
101	1.00	1.00	1.00	119
102	0.00	0.00	0.00	0
103	0.00	0.00	0.00	0
104	0.00	0.00	0.00	0
105	0.00	0.00	0.00	0
106	0.00	0.00	0.00	0
107	1.00	1.00	1.00	71
108	1.00	1.00	1.00	15
109	0.99	0.99	0.99	119
110	0.96	0.97	0.97	119
111	0.98	0.99	0.99	119
112	0.00	0.00	0.00	0
113	0.00	0.00	0.00	0
114	0.00	0.00	0.00	0
115	0.00	0.00	0.00	0
116	0.95	1.00	0.97	36
117	0.98	0.98	0.98	119
118	0.94	0.93	0.94	119
119	0.98	0.97	0.97	119
120	0.00	0.00	0.00	0
121	0.00	0.00	0.00	0
122	1.00	1.00	1.00	54
123	0.92	0.80	0.86	119
124	0.98	0.99	0.98	119
accuracy			0.98	

macro avg	0.97	0.97	0.97	10000
weighted avg	0.98	0.98	0.98	10000

Bibliography

[1] Xiao Yang and Jianling Sun. An analytical performance model of mapreduce. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 306–310, 2011.

[2] Shuo Wang, Yun Liang, and Wei Zhang. Flexcl: An analytical performance model for opencl workloads on flexible fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017 on*, page 27, 2017.

[3] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 45:1–45:12, New York, NY, USA, 2013. ACM.

[4] Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. Exascaling your library: Will your implementation meet your expectations? In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 165–175, 2015.

[5] *Extra-P – Automated Performance-Modeling Tool*. `www.scalasca.org/software/extra-p`.

[6] Juergen Schmidhuber. Deep learning in neural networks. *Neural Networks*, 61:85–117, 2015.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *neural information processing systems*, 141(5):1097–1105, 2012.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2015.

[9] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast multi-parameter performance modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–181, Sep. 2016.

[10] Patrick Reisert, Alexandru Calotoiu, Sergei Shudler, and Felix Wolf. Following the blind seer – creating better performance models using less information. In *European Conference on Parallel Processing*, pages 106–118, 2017.

[11] Simon S. Haykin. *Neural Networks: A Comprehensive Foundation*. 1998.

[12] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *arXiv: Neural and Evolutionary Computing*, 2017.

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. *Neurocomputing: foundations of research*, pages 673–695, 1988.

[14] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *ICLR 2015 : International Conference on Learning Representations 2015*, 2015.

[15] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[17] Martin Anthony and Peter L. Bartlett. Function learning from interpolation. *Combinatorics, Probability & Computing*, 9(3):213–225, 2000.

[18] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[19] Rodney R. Howell. On asymptotic notation with multiple variables. 2008.

[20] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[21] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[22] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[23] M. Kashif Ilyas, Alexandru Calotoiu, and Felix Wolf. Off-road performance modeling - how to deal with segmented data.

List of Figures

List of Tables

Listings